A Distributed Agent Architecture
for a Computer Virus Immune System

THESIS
Paul Kenneth Harmer, 1st Lt, USAF

AFIT/GCE/ENG/00M-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/00M-02

A Distributed Agent Architecture
for a Computer Virus Immune System

THESIS

Presented to the Faculty

School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Paul Kenneth Harmer, B.S. Electrical Engineering

1st Lt, USAF

March, 2000
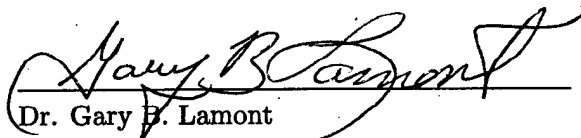
A Distributed Agent Architecture

for a Computer Virus Immune System

Paul Kenneth Harmer, B.S. Electrical Engineering

1st Lt, USAF

Approved:

Dr. Gary B. Lamont   _3 MARCH '00_
Dr. Gary B. Lamont   Date
Committee Chair

_____   _3 Mar 00_
Maj Scott A. DeLoach   Date
Committee Member

_____   _3 MAr 00_
Dr. Gregg H. Gunsch   Date
Committee Member

Table of Contents

List of Tables

AFIT/GCE/ENG/00M-02

Abstract

Information superiority is identified as an Air Force core competency and is recognized as a key enabler for the success of future missions. Information protection and information assurance are vital components required for achieving superiority in the Infosphere, but these goals are threatened by the exponential birth rate of new computer viruses. The increased global interconnectivity that is empowering advanced information systems is also increasing the spread of malicious code and current anti-virus solutions are quickly becoming overwhelmed by the burden of capturing and classifying new viral stains. To overcome this problem, a distributed computer virus immune system (CVIS) based on biological strategies is developed.

The biological immune system (BIS) offers a highly parallel defense-in-depth solution for detecting and eliminating foreign invaders. Each component of the BIS can be viewed as an autonomous agent. Only through the collective actions of this multi-agent system can non-self entities be detected and removed from the body. This research develops a model of the BIS and utilizes software agents to implement a CVIS.

The system design validates that agents are an effective methodology for the construction of an artificial immune system largely because the biological basis for the architecture can be described as a system of collaborating agents. The distributed agent architecture provides support for detection and management capabilities that are unavailable in current anti-virus solutions. However, the slow performance of the Java and the Java Shared Data Toolkit implementation indicate the need for a compiled language solution and the importance of understanding the performance issues in agent system design.

The detector agents are able to distinguish self from non-self within a probabilistic error rate that is tunable through the proper selection of system parameters. This research also shows that by fighting viruses using an immune system model, that known, and previously unknown, malicious code can be detected and removed from the system.

A Distributed Agent Architecture

for a Computer Virus Immune System

I. Introduction

The use of information systems has become vital to the mission of the Air Force, so much so that information superiority is identified as one of the Air Force core competencies. The Air Force is also quickly moving towards distributed Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C^4ISR) applications because information superiority is seen as the key factor to success in the 21st century. But, if the war-fighters are going to rely on information systems, then these systems must be robust, reliable, and secure.

While information dependence is increasing, the threat from malicious code, such as computer viruses, is also on the rise. The number of computer viruses has been increasing exponentially (Figure 1) from their first appearance in 1986 to over 45,000 different strains identified today (Sym00). Viruses were once spread by sharing disks. Now, with the rise of the Internet and global connectivity, malicious code is able to spread farther and faster than ever before.

Current anti-virus (AV) solutions are "reactive." They rely upon collecting and analyzing specimens of new viruses in order to update AV programs with the means of detection. This approach results in a slow reaction time to new threats and is quickly becoming too much of a burden to update with the increasing number of new viruses discovered each day. In the past, scan string updates were provided every two to three months; currently, vendors provide updates every few hours (Leo00). **To overcome this problem, a self-adaptive computer virus immune system (CVIS) based on biological strategies is developed.**

1
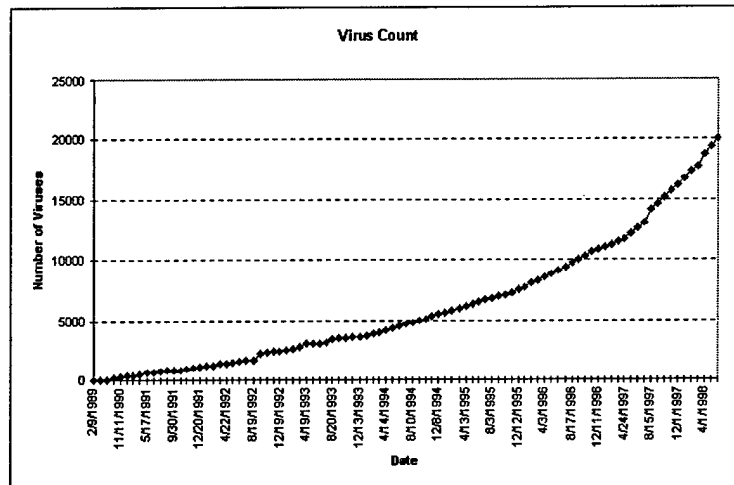
Figure 1.   Exponential growth in virus numbers (Net98).

1.1 Artificial Immune Systems

There are several computational techniques based on biological models including neural networks, evolutionary algorithms, and artificial immune systems (AISs) or immunological computation (Das99). The human immune system has been the target of considerable research interest in the medical community from which several theories of system behavior have been developed. The use of these immune system models to solve the computer virus problem has been suggested by (KA94, Das99, KSSW97, FHS97, LMV99). Immunological computation has also been applied to other problem domains, not all of which are in the computer security field. Some of the more interesting examples include decision support systems (Das98), multi-optimization problems (MTF96), anomaly detection in time series data (Das99), fault diagnosis (Das99), robust scheduling (HRN98), and loan application fraud detection (Her96). All of these utilize the pattern matching and "learning" mechanisms of the immune system model to perform core system features. A lot of theoretical groundwork in immunological computation has been completed, but only a handful of artificial immune systems have been built (HC96, HF99, Her96), none of which have been applied to the computer virus problem domain due to the inherent complexity of AV systems.

1.2 Software Agents

The highly parallel and distributed structure of the biological immune system suggests that an integrated architecture can be viewed as a multi-agent system (MAS), where separate functions are carried out by individual agents (Das98). Furthermore, the general immune system features represent a model of adaptive processes at the local level, with useful behavior emerging at the global level (Das99). This is similar to the description of MAS operations by the artificial intelligence community (DeL99a).

Despite all the research into software agents, researchers do not seem to agree on the exact definition of an agent (FG96). One definition speaks of agents as software components that communicate with their peers by exchanging messages (GK95). This effort focuses on inter-agent communication. Another paper on mobile agents uses the definition of a software agent as a program that can halt execution, ship itself to another computer on the network, and resume operation at the new computer (Ven97). It appears that the definition of an agent needs to be agreed upon up front in a discussion in order to communicate effectively therein. Therefore, the definition of an agent changes from paper to paper. The broadest definition includes anything that can be viewed as perceiving its environment and acting upon that environment (RN95).

Most of the current research is related to embodying agents with one or more of the qualities of agenthood. But, to solve complex problems, groups of agents must work together, often in heterogeneous environments (DeL99a). This creates engineering challenges as agents must now communicate, coordinate, and collaborate.

Along with the theoretical work on agents, several libraries, frameworks, or agent development kits (ADKs) have been developed (Ret99). Many of the initial ADKs were developed in scripting languages, such as TCL or Telescript, to facilitate operation in heterogeneous environments. But, with its network-centricity, sandbox security model, and platform independence, Java has become the environment of choice for the development of agent-based tools (Som97). The problem with these interpreted languages is speed, and very little research has been done on on the effects of this on high-performance computing

3

applications or highly combinatoric problems. Software agents need to be designed and deployed effectively in order to counter the AV combinatorics.

1.3 Identifying Viruses

The most common method of identify viruses is signature strings. A 16 byte string has become the defacto AV industry standard. Researchers at IBM have shown that 16 bytes is sufficient enough to identify malicious code with a 0.5% false positive rate (KA94). The largest problem with creating a new CVIS is the generation of these signature strings, or antibodies in an immune system. The problem is that only some of the $256^{16} = 3.4 \times 10^{38}$ combinations identify one or more valid viruses. Furthermore, if a valid string could be generated each microsecond, it would take a serial computer 1.08×10^{25} years to generate them all. This methodology uses simple exhaustive search, but even the generation of strings through machine learning techniques has been shown to be highly combinatoric (CO99). In general, the generation of strings and then testing their capabilities as virus identifiers is similar to the Boolean satisfiability NP Complete problem (NPc99). In this problem, a Boolean function is known (for example a function that describes one or more viruses) and the goal is to find the instantiation of function variables that returns a true value from the function. There may be one, many, or even no valid variable assignments that return true. The problem then degenerates into enumerating all possibilities, which leads to its classification as NP. This indicates that a polynomial-time algorithm does not exist for generating antibodies and so an approximation algorithm is the only choice.

Another issue with generating all possible scan strings is retaining them in memory or offline storage. Again, if 16 byte strings are used, storing all of the signatures would take $\frac{256^{16} \times 16 bytes \times 1 Mbyte}{10^6 bytes} = 5.4 \times 10^{33} Mbytes$. Even with the removal of known invalid combinations, this is far too large for current storage methods.

A parallel implementation of an antibody generation program is needed to reduce the high combinatoric burden and make the system practical. The large storage requirements can be reduced through creative methods, such as compression, but the sizes required also indicate the need for a distributed system. Finally, the increased connectivity of networked systems, which has aided virus spread in the past, can be used as a defensive weapon with a

distributed CVIS. This not only provides a parallel framework for the antibody generators and a distributed file system for their storage, but also provides the capabilities to eliminate intruders as they enter the system. A collective anti-viral self defense organization is created by delivering improved inoculations across the entire system.

1.4   Problem Statement

Current anti-virus methods are reactive. They rely on captured viral fragments in order to find future infections. This requires manpower intensive activities to capture, dissect, and produce signatures from the exponentially increasing numbers of new viruses. The inoculation of local systems with new signature updates, which are vital to the continued effectiveness of AV suites, often must be completed manually by the end user. The further use of the scanner also relies on the regular diligence of a system user. All of these factors result in AV solutions that cannot react to a rapidly increasing number of known viruses even if the product is installed and employed correctly. An AV methodology based on the biological immune system model is developed in order to create an autonomous and self-adaptive computer virus immune system.

An artificial immune system requires a diverse set of antibodies in order to detect harmful intrusions. Early work in this field has shown that the creation of viable antibody scan strings can be exponentially combinatoric. This research proposes efficiently decomposing the workload across a distributed set of nodes by utilizing parallel software engineering principles. This decomposition is done through the use of software agents.

The human immune system can be viewed as a multi-agent architecture whose entities each have their own limited capabilities. Total system functionality is the result of all these components working together to eliminate foreign invaders. In the information system domain, software agents are proposed as a viable methodology for artificial immune system construction. A fully effective agent-based CVIS will result from the collaboration between multiple software agents. Note that no such system has been completely designed or built.

5

1.5 Research Approach and Objectives

The goal of this research is to develop a agent-based CVIS. The approach to this problem is the integration of multi-agent systems, human immunology, computer immunology, parallel & distributed computation, and computer virus detection into an effective, efficient, and scaleable CVIS (Figure 2[1]). This approach is captured in the following research objectives:
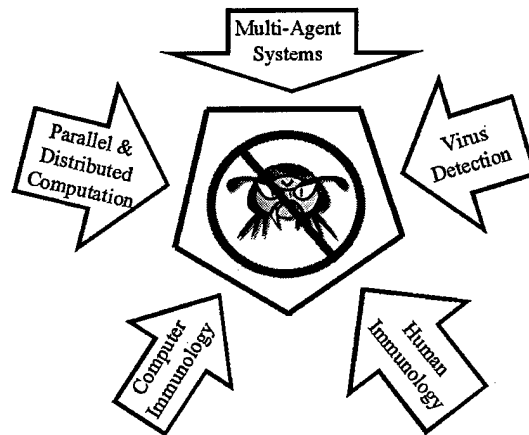
Figure 2.   Research integration domains.

Objective - Model Development Understand the capabilities and limitations of the immune system model as applied to the computer virus problem.

Objective - Matching Function The biological immune system utilizes imperfect matching for the recognition of foreign threats. A similar pattern matching function for the CVIS is desired that provides a relative similarity as opposed to a Boolean match/no-match condition. This objective is the investigation of pattern matching approaches to artificial immune systems.

Objective - Construct a Computer Virus Immune System Design, implement, and test a functioning CVIS based on distributed software agents. The agents will utilize parallel and distributed computing methodologies to reduce the combinatoric burden of the core CVIS processes.

---

[1]The "Boris" virus character is a trademark of Dr. Solomon's Software Limited.

6

Objective - Efficiency, Effectiveness, and Scaleability The system must be able to effectively detect, identify, and eliminate malicious code. These operations should be performed efficiently so that they may run unobtrusively in the background. Additionally, the distributed system should be expandable in order to increase its coverage and capabilities.

1.6 Scope

Viruses can attack files and specific disk sectors. Additionally, invaders can enter the system through various means including disk swapping, email, or file transfers (Nat99). This research is focusing solely at detecting file infections. A design goal is to modularize detector agents so that they could easily be extended in the future for the scanning of other input sources.

The major problem with a CVIS is the generation of antibody pattern matching strings. Many methods have been suggested, including the use of random search (HF99), genetic algorithms (FSJP93), and machine learning (CO99). All of these have been shown to be combinatoric and computationally intensive. Without *a priori* knowledge of the large self and non-self space, any methodology of searching for non-self strings and providing a diverse coverage is not likely to out perform a random search (WM97). Due to a lack of a self or non-self space characterization, this project only uses the random generation of strings for antibody creation.

Much of the current research on software agents targets their specification, construction, communication, and attributes (BB98, Bra97, Bre98, Cha97, Ret99, HCK95, GK95, FG96). The focus of this project is not on generic agent system development research, but on the possible use of agents as an effective means to construct a CVIS application.

Finally, the purpose of this research is not to create an accurate representation of the human immune system, nor to prove the validity of it. The goal is to use a model of the functions and features of the biological immune system for an improved anti-virus information system.

1.7 Thesis Overview

This document provides a discussion of this research effort, starting with a literature review of viruses, virus detection methods, biological immune systems, immunological computation, and distributed agents in Chapter Two. Chapter Three and Four provide a discussion of the system design methodology. Chapter Five presents the results of system testing. Finally, Chapter Six concludes that distributed computer immunology could provide a viable computer virus detection and elimination system. This chapter also discusses avenues of future research within the areas of immunological computation, computer security, and virus detection.

II. Literature Review

This chapter presents the background knowledge required to understand the development of an agent-based artificial immune system for combating computer viruses. Each section gives an overview on one of the aspects of this multidimensional integration problem. Section One covers computer viruses, the many types, and current anti-virus methodologies. The next section gives an overview of the human immune system, its functions, and key components. This is followed by a discussion of the current research into artificial immune systems. The final section contains a summary of agents, their design, and their applicability to this problem domain.

2.1 Computer Viruses

The term computer virus is often attached to unwanted code that does malicious activities on its host computer. Applying the term in this fashion is imprecise and misleading as viruses are actually only one form of "rogue code." Malicious code can take the form of Trojan horses, worms, or viruses. Additionally each of these may or may not contain a payload or "logic bomb." It is the payload routine that actually performs the "damage."

A Trojan horse is a program the masquerades as one program, while it actually performs an entirely different task altogether (Ska96). This variant gets its name from the original wooden horse used to conquer the city of Troy (Ska96). A typical Trojan is given the same name as another program and waits for an unsuspecting user to execute it by mistake. An increasingly common application of Trojans is to use them to bypass computer security features on behalf of an intruder in order to gain unauthorized access to a computer system. A well known example is the Cult of the Dead Cow's Back Orifice.

Worms are programs which execute independently and utilize a computer network in order to propagate themselves (Coh94, Hof90). They often take advantage of security or communications protocol loopholes in order to spread (Ska96). The first worms were built at the Zerox PARC research center. They were designed to perform useful work in a distributed environment, such as finding idle resources (Hof90). These original worms would probably be called mobile agents today. The Melissa virus is more accurately termed

9

a worm as it used the features of Microsoft Exchange e-mail in order to spread itself across networks.

A computer virus is a program that can "infect" other programs by modifying them to include a, possibly evolved, version of itself (Coh94). One distinguishing feature of viruses is that they are parasitic. They require a host to run them and to spread their viral code (Hof90). This is usually another executable program although other hosts, such as disk boot sectors, are also targets.

A sequence of symbols is an element of a "viral set" V if, when a machine interprets that sequence of symbols v, it causes some other element of that viral set, v', to appear elsewhere in the system at a later point in time. A viral set most commonly makes an exact copy of itself on the same machine. However, the formal definition is much broader than that because any sequence of symbols that is interpreted on a machine could potentially contain a virus (Coh94). Finally, as stated in Cohen's definition, viruses can evolve through a countably infinite number of different variations. Therefore, there exists the possibility of an infinite number of different viruses. This fact makes the problem of detecting viruses intractable (Coh94).

The problem of creating a Turing Machine program capable of determining, in finite time, whether or not a given sequence of symbols is a virus is known as the "decidability" issue (Coh94). Cohen proved in his PhD dissertation that given machine M and a nonempty set V, the question as to whether or not the pair (M, V) is a viral set is undecidable. The proof is a result of the halting problem (Coh94). Due to the intractability of detecting viral code, the search space has been reduced by classifying viruses and then applying deterministic methods of detection (such as pattern matching) to each class.

2.1.1 Taxonomy. The features and classifications used to catagorize computer viruses are usually discussed in terms related to Microsoft DOS-based computers and their variants (Windows 3.1, Windows 95/98). The generic nature of Cohen's definition of a virus correctly implies that they can be written for any computer system (Int98). However, personal computers running DOS operating systems are the host for the largest number of computer viruses. In a collection of 31,774 specimens, 81.43% targeted DOS/Windows

10

Table 1. Common virus strains and their classification (Sym00).

Name	Type	Resident	Stealth	Encrypting	Polymorphic
Cascade	File	√		√	
Compiler.1	Companion		√		
AntiEXE	Boot	√	√		
SatanBug.Natas	Multipartie	√	√	√	√
WM.Concept	Macro	√			

as the host platform ((Ref99) as of 10 Jan 2000). This is due to the large population of such systems and the lack of security inherent in their design. For instance, there are only 5 viruses that target Linux machines, an operating system that enforces access rights on files and system resources (Ref99).

Computer viruses are usually classified by their method of infection. The common subclasses of viruses are file infector, boot sector, and macro viruses. Within each of these genera, there also exists subcategories. Additionally, many viruses contain advanced features or capabilities which distinguish them from simple varieties (Table 1).

2.1.1.1 File Infector. The file infector is the type most commonly associated with the term computer virus. File infection viruses work by inserting their code into executable files, just as the biological virus works by inserting its DNA code into living cells (Ska96). The host file then executes the malicious code on behalf of the virus. If the virus attaches to a non-executable file, such as a text file, it may corrupt some data, but it can not be executed, and so it won't reproduce (Lud96). On a DOS based machine, executable files include COM, EXE, and also some more esoteric types such as SYS and OVL. Any executable code is a potential target, even shared libraries such as DLLs. Often file infectors target a specific file. These viruses make use of a detailed knowledge of the files they attack in order to hide within the target file's structure (Lud96).

The virus code itself can either be appended, inserted, or even interleaved into the host file. The crudest method is to simply overwrite the existing file with virus code. This results in no change to the original file length, but the original code no longer operates correctly, and so the presence of a virus is quickly recognized. The most advanced method

11

is to interleave viral code into unused portions of the original file. This requires advanced flow control instructions to ensure that the viral code is executed in the correct order. This method also results in no change to the original file length. The most common infection places a small virus subroutine at the end of a file and then inserts a call to that routine just ahead of the original file's code (Figure 3). Some commonly known examples of file infectors include Cascade, Green Caterpillar, and Jerusalem.

Jump Instruction	Host File Code	Viral Code
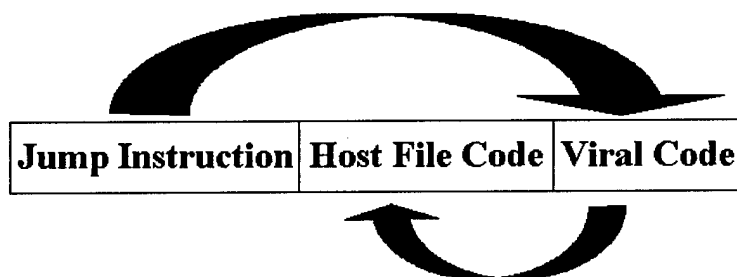
Figure 3.   File infector virus methodology (Sla96).

2.1.1.2 Companion.   The companion virus is more accurately classified as a Trojan horse. It masquerades as a legitimate program and takes advantage of the operating system methodology to run the virus instead of the program the user thought they were executing. In the MSDOS operating system, directly executable files have only three possible extensions, COM, EXE, or BAT. If a user enters a program name at the command line without an extension, COM is assumed. If *myprog.com* is not found, then EXE is assumed. If *myprog.exe* is not found then BAT is assumed. This pattern is continued through the current directory and then on each directory specified in the PATH environment variable (Ska96). A UNIX file system operates similarly but only searches for *myprog* in the current path.

The companion Trojan/virus uses this program search feature to its advantage. First, it finds an EXE or a BAT file to infect. But, instead of infecting the file itself, it simply places a copy of its viral code in a file with the same name as the target, but with a COM suffix. A COM file can be "infected" by placing a copy of the virus with the same filename as the COM file, but in a directory that appears earlier in the path list.

Companion viruses typically do not spread well. They are easily identified due to the fact that multiple programs with the same name are appearing all over a user's directories. Additionally, it is difficult for them to spread to other computers. This is because getting them there would generally require copying the Trojan instead of the actual program to a disk (Ska96). Compiler and its variants are examples of this virus type.

2.1.1.3 Boot. Boot viruses come in two major varieties, boot sector and partition sector. Both types attach themselves to specific areas of a disk that are loaded and executed on startup. The boot sector virus attacks a specific location on a computer's disk, known as the boot sector or the master boot record (MBR), while the partition sector virus attaches itself to the partition record (Lud96).

A physical hard disk can be divided into several logical disks. The table that maps the partitions, as well as a small program to complete the translations, is stored in the partition record or sector. On boot up, the small program is read from the partition sector, checks for a valid partition table, and then proceeds to load the boot sector from the designated active logical disk (Ska96). The boot sector contains the first system program a computer loads from disk into memory and executes. This is normally reserved for the computer's operating system loader.

A partition sector virus infects a hard disk's partition sector (Ska96). But, this sector is usually limited to 512 bytes in size. Additionally, without the partition table and the translation program, the disk does not operate properly. So, an effective partition virus must copy the correct partition sector information to somewhere else on the disk and then pass control to it after it has completed its viral code. A simple overwriting partition sector virus would render the hard disk inoperable (Ska96).

On startup, the boot sector is loaded from disk and executed. By placing its viral code into the boot sector of the disk, a virus can gain control of the computer immediately upon boot-up (Figure 4). This allows the virus to execute before anything can detect its existence (Lud96). If a computer is booted from an infected floppy, the virus can immediately spread to the hard disk or any other disk attached to the system. If there are no other disks present, then the virus does not have any additional hosts to spread

13

to. Many boot viruses solve this problem by becoming memory resident (Section 2.1.2.1) where they wait for new hosts to arrive. Due to the success of this infection method, many examples of this type exist. Some of the most prevalent viruses of this type include Form, AntiExe, and EmpireMonkey.
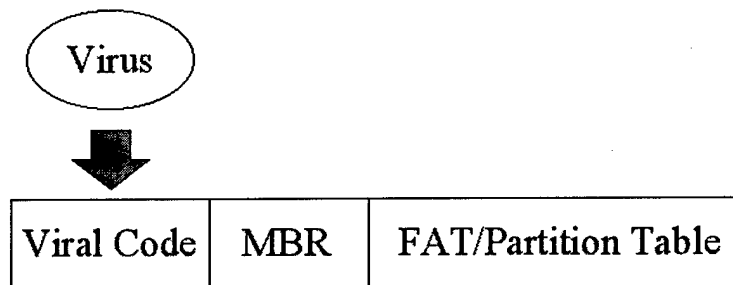
Figure 4.   Boot sector virus methodology.

2.1.1.4   Macro.   The macro virus is the latest type to enter the taxonomy. They are unique in that they infect what had previously been impossible, data files. The macro virus is a section of code contained within an application document. The most common today are MSWord macro viruses, although examples exist that infect the documents created by any of the MSOffice applications as well as others, such as WordPerfect documents. In Microsoft's desire to make the MSOffice family of products more powerful, by adding Visual Basic for Applications (VBA), they have opened the door for executable code to be contained within data files. The intent of this capability was to add automation capabilities to otherwise static documents. As a further boon to virus writers, macro viruses are much easier to write than before because macros use high level languages and do not require specific operating system knowledge.

The idea of the macro virus was first proposed in 1989 (For97). The first macro virus to appear in the wild, Concept, was discovered in 1995 and since then, this type has risen to be the most successful infector. Concept continues to grow more rapidly than any other previous virus in history (Nat97). Boot viruses usually travel via diskette, while the macro virus is spread via email (Nat97). In fact, macro viruses can travel via all media making them highly contagious. MSWord macros also work on MS Word for Macintosh (Wel96). In the MSOffice macro exists the capability to infect multiple platforms, multiple operating

systems, and travel via all mediums. The rise of the macro virus has mirrored the rise of the Internet and the speed and international reach of email continues to contribute to the rapid spread of new and old macro viruses (Nat97). Although Concept was the first, many more have appeared, including WM.Wazzo that alters MSWord files and XM.Laroux that infects MSExcel worksheets.

2.1.1.5 Multiparatite. A multiparatite, or multipartite, virus is the classification used when a virus employs two or more infection methods (Ska96). The most common is for a virus to be both a file and a boot infector, although any combination of two or more methods is allowed under this category. The boot-file infector loads its code from the boot sector immediately upon system startup. It then goes resident and proceeds to infect files in the user's system. Executing an infected file typically loads the virus onto the boot sector if it is not already there before infecting other targets. SatanBug.Natas and Anthrax are examples of this viral type.

2.1.2 Advanced Features. The virus advanced features have evolved over time as virus writers attempt to circumvent the virus hunters. The goal of these concepts is to make the virus spread more effectively and avoid detection for as long as possible.

2.1.2.1 Resident. A resident program executes and then remains in memory. It is then executed at regular intervals or by some event, such as a disk being inserted into a floppy drive. This is known as terminate and stay resident (TSR) in the MSDOS operating system. While resident, the program can watch for disk or file operations, which it then piggy backs its infection routine upon (Ska96). For instance, a file infector virus may intercept all file access routines and append its code to every file a user opens.

2.1.2.2 Stealth. Stealth is an active defense by a virus to avoid detection (Ska96). This can be accomplished in many ways, but usually involves intercepting operating system calls and redirecting them to decoy data. The Brain virus did this by overwriting the disk boot sector, but not before placing a copy of the original data elsewhere on the disk. The key to its stealth was that it intercepted any calls to look at

15

the boot sector and redirected them to the copy of the original code, thereby hiding its infection.

2.1.2.3 Encryption. The common method of detecting virus infections is through the use of a scan string. In order to avoid the pattern match, a virus needs to defeat the appearance check of the anti-virus scan. The self-encrypting virus does this by encrypting its code and encrypting it with a different key on each replication (Ska96). This way, the virus does not have a consistent appearance in all instances of the virus code and so a single string is ineffective. Additionally, storing all possible combinations of the string is prohibitive to the scanner. However, the encryption/decryption routine could be used as the scan string since altering its code would render the virus inoperable. In response to this threat from the anti-virus scanner, the polymorphic virus evolved.

2.1.2.4 Polymorphism. The polymorphic virus alters its actual program code on each infection. The most common method is to insert assembly language routines that do not alter program execution, but which change the program appearance. This renders a generic scan string ineffective. The simplest method is to insert no operation or NOP commands. These are legitimate processor instructions that do no useful work. Therefore, a series of NOPs can be inserted, or interleaved, between actual program instructions to alter the code's appearance, but not its functionality. Another method is to accomplish the same task through the use of alternate programming instructions. Each of the following commands sets an internal register to zero (Ska96).

```
mov AX, 0    ;Load the AX register with 0
xor AX, AX   ;Set the AX register to 0 logically
sub AX, AX   ;Subtract AX from itself
```

More advanced polymorphism engines are able to alter sequences of instructions, yet perform the same functionality. The combinatorics of trying to find all possible combinations can render scan strings completely unusable for virus detection. There are still many scanners which are unable to detect V2P6 - one of the first extremely polymorphic viruses (Bon94).

2.1.3 Detection and Elimination. If a virus is spotted, it is usually quite easy to remove (Ska96). This has led virus programmers towards ingenious methods of infection, stealth, and camouflage. As a response, the anti-virus community has produced more complete and robust detection routines. But, however advanced, there remains only four basic ways to detect a virus: appearance, behavior, change, and bait (Ska96).

2.1.3.1 Appearance. Detection by appearance is the most common method employed. This technique involves pattern matching between a suspected infection and a scan string. This scan string is a captured piece of known viral code which, hopefully, uniquely identifies the virus. A short string is more general and also can be compared faster. However, a longer string would reduce false positives. A balance on string length must be reached. 16 bytes has become the *de facto* anti-virus industry standard (KA94).

The problem with scan strings is that they are passive and reactive. They can only identify viruses after the infection has occurred. Also, they can only detect known viruses for which signatures have been extracted. New viruses, or even new strains of known viruses, are left undetected. This requires constant updates to the signature database in order to keep up with the high viral birth rate (Ska96). On a positive note, because the string is from a known virus, a detection is also an identification. This knowledge can be used to repair the infected file based on the virus characteristics, instead of just deleting it.

2.1.3.2 Behavior. All viruses must make copies of themselves and this is usually accomplished through low level operating system routines. These routines can be intercepted by the virus checker in order to monitor virus-like activity. The problem is that legitimate programs also need to perform these operations, such as writing to a file. The behavior scanner needs to be a bit more discerning by looking for more dubious actions, such as trying to write to an executable file. Virus-like behavior detection is also known as heuristics (Int98).

Detection by behavior is an active defense. The anti-virus system is able to detect viral activity as it is occurring and intercept it before any damage results. The problem

with this method is that the scanner must always be running. This leads to overhead and system slow down. Additionally, intercepting system calls interferes with the normal operating system operation and can also interfere with user programs.

2.1.3.3 Change.   In order to insert a copy of itself into a host, a virus must alter something. This change is exactly what is looked for in change detection. In order to detect a change, the initial state, file size, file date, or checksum must be known (Ska96). It is easy for a virus to revert the last change date, or misreport the file size via stealth, but a checksum is almost impossible to subvert. The scanner typically uses a cyclic redundancy code (CRC) to add up all the bytes in a file an produce a single number. The odds of two different programs summing to the same result are so remote as to make this an effective means of change detection (Ska96).

The problem with change detection is that capturing the initial state can be a somewhat lengthy and laborious process (Ska96). All of this data must then be placed in a secure database somewhere and the database must be updated when any new programs are installed. This database can then become a target of corruption, spoofing, or infection by a enterprising virus. Also, it can be difficult to determine if the uninfected state was actually captured. If a virus sneaks in before the known safe state is captured, the infected file becomes the uninfected state to the change detection program. The virus has slipped through the defenses. Finally, change detection is useless on files which change often, such as data files.

2.1.3.4 Bait.   Detection by bait is a variant of detection by change that avoids having to capture the unchanged state of all programs in the system. In this method, a dummy file is allowed to be a sacrificial lamb for a lurking virus. The uninfected state of the dummy file is known and saved. This file is then checked regularly, or performs a self check, to see if it has been infected. The attractiveness of the bait can be enhanced by placing it in likely virus targeted directories, such as with the system files, or by using attractive naming conventions. Additionally, the dummy file could actually perform actions which attract viruses, such as executing often or performing many disk accesses (Ska96).

This method has several advantages, but one big disadvantage. Because the program is known, there is almost no chance of a false positive detection and the virus code can be easily isolated from the original program structure. Therefore, the exact virus signature can be automatically extracted and used as a scan string (KSSW97, KA94). The problem with this method is that there is no guarantee that the bait will be taken. The file can wait for long periods of time without any infections, while all the files around it are targeted.

2.1.4 Biological Connection. The term "computer virus" was coined by Fred Cohen and takes its name from the entity in the biological world known for making humans sick (Ska96). No one argues that the computer virus is alive, but ironically biologists argue whether or not their viruses are actually alive as well (Ska96). Alive or not, the computer virus exhibits some striking similarities to simple forms of life, such as one celled organisms, to make the comparison valid. Clearly, the similarities between life and the computer virus exist, or Cohen would not have used the term initially.

Biological entities multiply by replicating their DNA. The DNA contains all the definition of self and the instructions to carry out the cell's processes. The computer virus' DNA is its program code. The computer virus replicates by copying its program code. Interestingly, the information contained in the DNA of a typical biological virus, such as polio, would equal about 5K, the smallest biological virus would only be about 200 bytes (Ska96). In a computer virus collection totaling 17,743 different specimens, 79% were less than 5K in size (Table 2). Most, 44%, were less than 1K (Ref99). The smallest amount of code to construct a crude overwrite virus is 25 bytes.

Computer viruses target a specific platform (DOS, Mac, etc.) and usually target a specific program type (EXE, COM, etc.). The biological virus also targets a specific species. For example, Ebola Reston only targets monkeys, while Ebola Zaire is deadly to humans (Pre94). Also, certain biological attacks only target specific types of cells. Rhinoviruses, which cause the common cold, attack the membranes in the nose, throat, and sometimes the lungs (Fou93).

When a computer gets "ill," it does not necessarily need to show any obvious signs of infection. Likewise, an infected cell can go on living without any noticeable signs. The

Table 2.   Viral length statistics (Ref99).

Size (Kbytes)	Number	Percentage
Less than 1	7735	43.59%
1 - 5	6360	35.85%
5 - 10	1365	7.69%
10 - 25	1422	8.01%
25 - 50	537	3.03%
50 - 75	198	1.12%
75 - 100	44	0.25%
100 - 150	34	0.19%
150 - 200	17	0.10%
200 - 250	7	0.04%
250 - 300	14	0.08%
300+	49	0.28%

most famous example would be Typhoid Mary, who passed on Typhoid Fever to thousands without showing any signs of the disease herself.

Many biological diseases have a certain incubation period in which they lie in wait until unloading their payload sometime in the future. The computer "disease" may also wait until a specific period has passed, showing no signs of infection. This may be a date, like the Michelango virus, or simply an event driven countdown clock (Bon94).

Simple life reproduces through replication. Mathematically speaking, it makes multiple copies of itself. This, in turn, leads to exponential population growth (Bac96), which can be seen in the growth of simple bacteria and also in the computer virus population numbers.

Because computer viruses exhibit so many parallels with simple forms of life, it only makes sense to suppress them in a manner similar to the methodologies used by higher forms of life. This requires an understanding of the major functions and structures used by the human immune system to counter infection. An abstract view of these processes can then be used in a model for the operation of a computer virus immune system.

2.2 Human Immune System

The human immune system provides a defense in depth framework for the protection of internal resources against foreign invaders. This multi-layered defense is made up of many individual components working together towards the common goal of protecting the body against these pathogens (Figure 5). The immune system consists of three defense levels, local barriers, inflammation (or non-specific response), and specific response (BP78). All levels work together to form a complete defense with one or all of them being initiated after the recognition of an antigen (BP78).
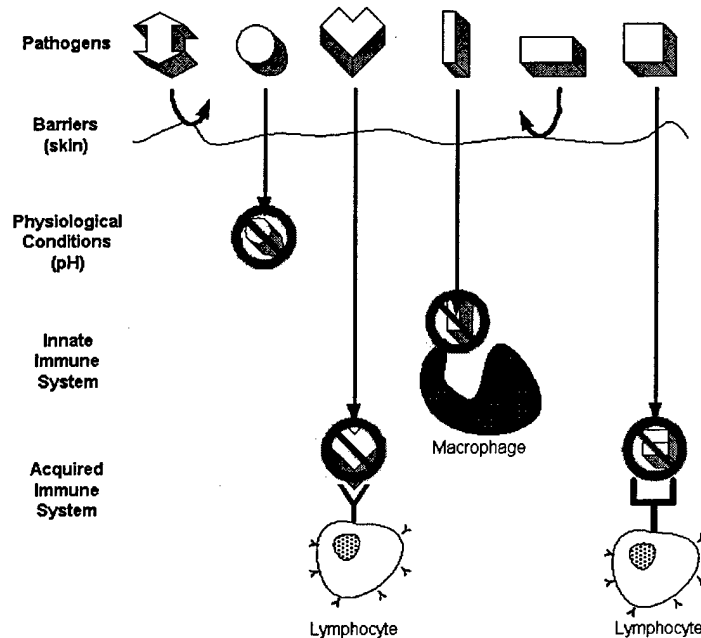
Figure 5.   The immune system multi-layered defense (Hof97).

An *antigen* is any substance that can stimulate the immune system (BP78). The most common antigens are proteins, such as those found in bacteria and viruses (Gor93). These molecules have multiple surface reaction sites, or antigen determinant sites, that act as interaction points for the immune system cells (BP78).

The human immune system has the innate capability to recognize cells and proteins that compose the body's systems. When cells or protein substances that are not compo-nents of the original system are introduced, they are recognized as non-self and a defense is

21

initiated (BP78). The essence of immunology is understanding how the body distinguishes "self" from "non-self" (BSL96).

Immunity refers to all the mechanisms used by the body as protections against agents that are foreign to the body (BSL96). The mechanisms operate in solution or within cells to provide humoral and cellular immunity. These are further divided into innate and acquired immunity (Figure 6).
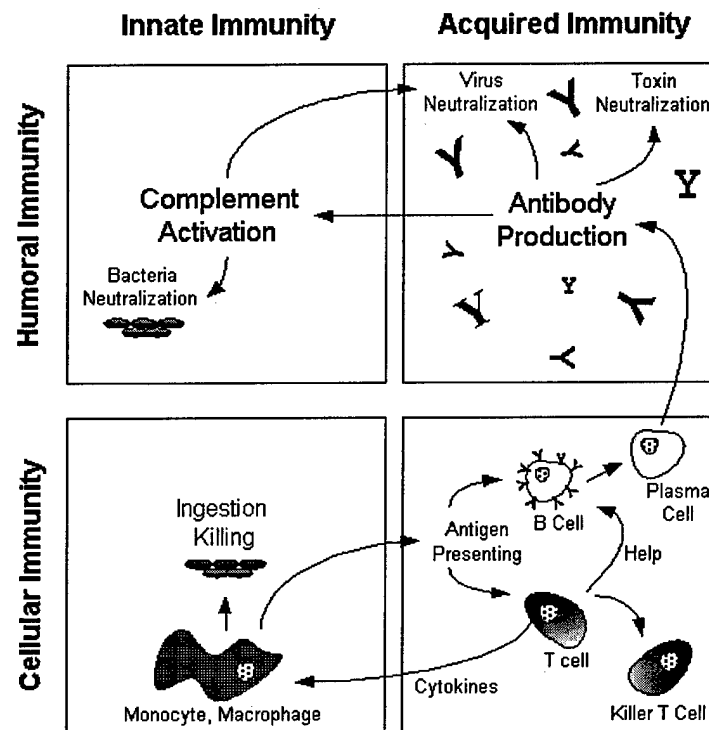
Figure 6.   Immunity type relationships (Gic98).

2.2.1 Innate Immunity.   The innate immune system is made up of all the elements which an individual is born with and that are immediately available to recognize and destroy foreign invaders (BSL96, Jan93). The innate immune system has the power to destroy many pathogens on first encounter (Jan93). This subsystem includes barrier elements such as the skin, mucous membranes, and the cough reflex, but there are also internal components. These include as vast array of serum proteins and specialized cells.

An important part of innate immunity are the phagocytic cells. Phagocytes are amoeba-like cells that engulf and digest microbes, such as the macrophages (BSL96). The macrophages also signal other immune system reactions. Another part of the innate response is a class of blood proteins known as complement. One type of complement protein, when chemically stimulated, can bind to any other protein - those on bacteria as well as those on our own cells. This binding triggers the activity of other complement molecules which in turn attract the phagocytes (Jan93). The complement itself can also destroy cells without the services of the phagocytes by punching holes in the cellular membranes of invaders. This causes water to rush in and explode the cell (Jan93). Unlike microbes, our own cells are equipped with proteins that inactive complement, preventing the body from attacking self.

All of the innate immune system components either affect pathogenic invaders directly or enhance the effectiveness of other immune system reactions to them (BSL96). However effective the innate immune system is, it cannot protect against all infections. Microbes evolve rapidly, which enables them to devise methods to evade the innate capabilities of the immune system. To overcome this, vertebrates have developed an adaptive immunity.

2.2.2 Acquired Immunity. Acquired immunity enables the body to recognize and respond to any microbe, even if it has never faced the invader before (Jan93). There are two types of acquired immunity, cell mediated and humoral mediated. The cell mediated response is orchestrated by the T cells. This subsystem combats fungi and most viruses. The humoral subsystem is organized by the B cells. They are effective against bacteria and some viruses. Both areas cooperate with the innate immunity components in the final defense of the body (BP78). The work horses of the innate and acquired immune systems are the white blood cells.

2.2.3 Cellular Immunity. There are six forms of white blood cells that all derive from a single stem cell in the bone marrow. The first four cells offer a nonspecific response to infections. Neutrophils immediately engulf bacteria on contact and also send out early warning signals. Monocytes evolve into macrophages that engulf antigen. Eosinophils

23

attack various parasites. Finally, basophils contain granules of histamines and other chemicals related to allergies (Hal98). The other two are lymphocytes that offer immunity tailored to a specific antigen. Lymphocytes are covered with sensitive cell-surface receptors that are genetically programmed to recognize different antigen. This sensitive recognition capability allows these T cells and B cells to form a specific response to an antigen (Hal98).

2.2.3.1   T-Cells.   T cells are so named because they mature in the thymus (BP78). Stem cells are created in the bone marrow and migrate from there to the thymus. In the thymus, stem cells differentiate into immature T cells. The immature T cells are converted to immunocompentent T cells by maturation hormones (BP78). These antigen matching cells are then put through a censoring process before they are released to the body (Figure 7). During this probationary period, the immature T cells that attack self proteins are eliminated. After this censoring, known as *negative selection*, only non-self reacting T cells remain (BSL96). These mature, censored, T cells leave the thymus and travel both the circulatory and the lymphatic systems (BP78).
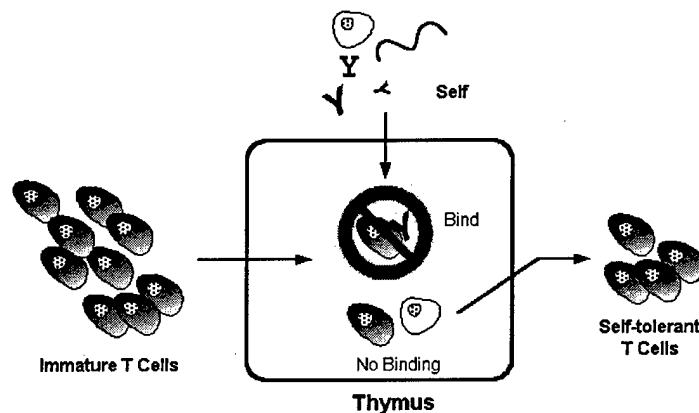
Figure 7.   T cell negative selection process.

T cells coordinate the entire immune response. They send signals that promote antibody formation and they also eliminate viruses that hide inside infected cells (Hal98). There are two subtypes of T cells, Helper, or CD4 T cells, and Killer, or CD8 cells (Hal98).

In the presence of antigen, T lymphocytes are capable of producing lymphokines. Lymphokines are mediator substances that facilitate the immune response and bring about

24

the destruction of the antigen. These substances attract other immune system cells, such as macrophages, keep them in the area, sensitize other cells, and induce the cloning of sensitized cells (BP78). These factors work to slowly build up the immune response.

After a Killer T cell recognizes an antigen on the outside of a cell, bubbles known as granules form inside the T cell. These granules then move to the side of the cell that binded with the antigen. In about 5-10 minutes, all the granules have migrated to the contact side and then break against the membrane of the infected cell. The granules contain perfoin, which perforates the target cell's membrane, rupturing and killing it (Hal98). Killer T cells must receive a set of signals from Helper T cells in order to remain alive and replicate. If not, they recognize that something is wrong and commit suicide (Hal98). So, any attack is short lived unless the proper chemical signals are received.

The Helper T cell is an immune system coordinator, but it must first be activated by binding to an antigen. After activation, Helpers generate many chemical signals. These have the ability to boost the number of lymphocytes by inducing cell cloning (Hal98). The Helper T cells also perform *costimulation*. B cells are activated by two signals, one occurs when its antibodies bind with the appropriate antigen, and the second is a validation signal from a Helper T cell that also binds to the antigen. Because the T cell was previously censored to only recognize non-self, this provides an assured confirmation of a valid pathogen (Hof97).

2.2.3.2 B-Cells. B cells are so named because they develop in the bone marrow (Hal98). They are studded with many "Y" shaped detectors called *antibodies*. Antibodies are also known as immunoglobin. Parts of the immunoglobin chains unfold and expose small patches, or clefts on their surface, which make them highly specific antigen binders (BSL96).

B cells are postulated to develop and specialize via *clonal selection*. In this process, there are two stages. In stage I, the stem cells develop into antigen reactive B cells. The B cells produce immunoglobin, which is presented on the cell surface. These cells then divide to produce clones that genealogically produce the same antibodies with the same specificity (BSL96). In stage II, the cells enter circulation. When they are activated by

antigen contact, they multiply to proliferate memory cells or they differentiate into *plasma cells*. The activated plasma cells produce free-floating antibodies at a rate of 10 million per hour (Hal98). In this way, only those B cells that match antigen are reproduced or stimulated to produce antibodies. This reaction can take 6 to 14 days on the first encounter. On the second occurrence, the secondary response is greater due to a higher concentration of memory cells already present (BSL96).

Working with clonal selection are two additional processes that aid the immune system in adapting to specific antigen. After activation, B cells can go through *hypermutation* (Figure 8). In this process, the cell reproduces itself with very high mutation rates. This creates daughter cells that are a little bit different than the parent and hopefully have a better binding affinity for the pathogen. The T cells are responsible for self tolerance, enforced through costimulation. This enables the B cells to hypermutate freely with the goal of adapting to specific antigen (Hof97).
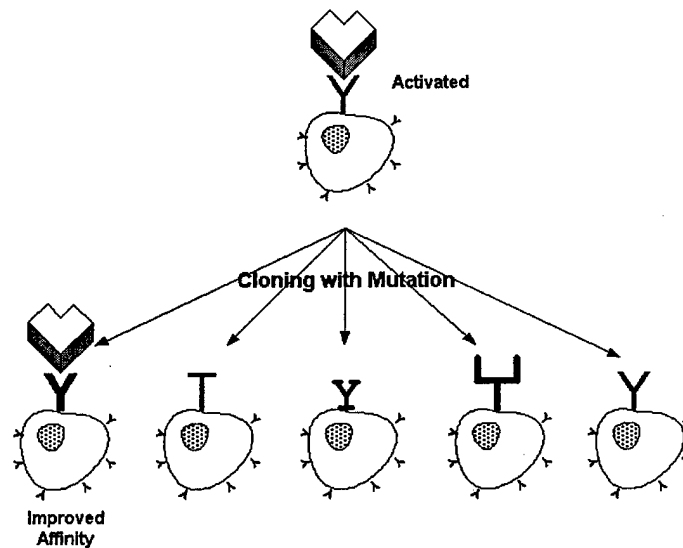
Figure 8.   Clonal selection with hypermutation (Hof97).

This works within a Darwinian selection process known as *affinity maturation*. Those B cells that better match the antigen, divide to produce more memory and plasma cells. Those B cells that do not match soon die. Through this process, the B cells remaining in

26

the system have a higher affinity for matching pathogen protein features and are better able to induce a secondary response on the next encounter.

2.2.3.3 Antigen Presenting Cells. The third type of cells involved with the acquired immune response are *Antigen Presenting Cells* (APCs), such as macrophages (BSL96). They are capable of collecting, either intentionally or because of infection, antigen and presenting it on the cell's surface for possible recognition by the T cells. The body's cells are continually building up and breaking down proteins. The APCs take in antigen, break them down into smaller fragments and then present these fragments on the cell's surface using MHC (BSL96).

T cells recognize small peptides, but only when they are presented by the *major histocompatibility complex* (MHC). The MHC is a set of markings on the surface of a cell that tag it as "self" and that also contain a groove for the capture and presentation of small protein chains, peptides (Hal98). Only when antigen are simultaneously presented by the piece of self in the MHC are the T cells stimulated. This requires the CD4 and CD8 receptors to bind with both the MHC and the foreign protein (Hal98).

There are two classes of MHC. Humans carry 6 out of at least 200 variants of class I MHC and 8 out of about 230 types of class II MHC (Hal98). Class I is recognized by the CD8 receptors on the Killer T cells. It is expressed by all cells, which provides the body with global coverage (Hal98). Class I MHC is able to capture antigenic peptides floating within the cell's cytoplasm and transport them to the cell membrane for presentation. The Class II MHC is recognized by the Helper T cell's CD4 receptors. It operates in a similar manner as class I varieties, but it is only present in certain APCs, such as macrophages (Hal98). The class II MHC captures and presents antigen that has been digested within an APC's vacuole. The combination of the both types of MHC work together to combat more aggressive invaders. If a pathogen is able to break free of a macrophage's vacuole, parts of it are still picked up by the class I MHC and presented. This gives specific APCs two lines of defense against infection.

27

2.2.4 Autoimmune Disease.   Autoimmune disease is the result of the components of the immune system perceiving self as antigenic, or non-self. The immune system then proceeds to destroy self as if it were foreign. Both T and B cells are implicated in this process (BP78).

Artificial immune systems must eliminate any type of autoimmune reaction in order to be an effective solution and provide a consistent operation that is trusted by the user. AISs have successfully fielded models of immune system operations and avoided autoimmune reactions to create biologically inspired information processing systems (Das99). In doing so, they have embodied the features and strengths of the immune system within a computation platform.

2.3 Artificial Immune Systems

There are several computational techniques based on biological models including neural networks, evolutionary algorithms, and artificial immune systems or immunological computation (Das99). The human immune system has been the target of considerable research interest in the medical community from which several theories of system behavior have been developed. The natural immune system has been shown to be a highly parallel and decentralized information processing system that can generate selective responses to foreign invasion (Das98). Immunological computation is working towards solving real-world problems by using this methodology.

The biological model has been interpreted differently by many researchers in its transformation to the computer algorithm domain. Most of these computational models emulate one or more of the functional components of the biological immune system. Immunological systems based on the idiotypic model mirror the functions of the B cells, while Forrest's negative selection algorithm (Figure 9) models the interactions of T cells (FAPC94). These biologically inspired algorithms have been applied to many problem domains including computer security, decision support systems (Das98), multi-optimization problems (MTF96), anomaly detection in time series data (Das99), and fault diagnosis (Das99). While the analogy between the immune system's biological processes and computer security or machine learning are fairly obvious, there are many differences between computers and biological

organisms which make the mapping difficult. The success of the immune system analogy in a computer virus immune system rests in mapping the biological processes to the correct level of abstraction to the algorithm domain (FAPC94).
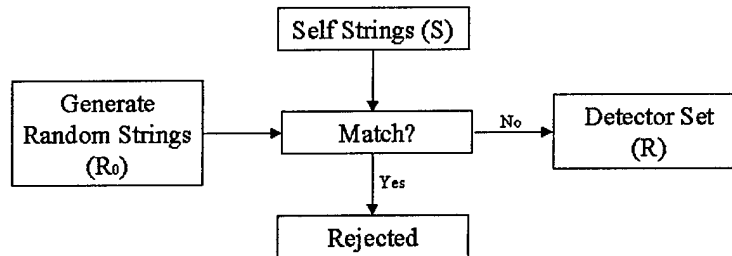
Figure 9.   Negative selection algorithm (FAPC94).

2.3.1   Negative Selection and Imperfect Matching.   Dr. Forrest's research group at the University of New Mexico (UNM) introduced the connection between the biological immune system and computer security in 1994 (SHF97). Their work has focused on the use of negative selection to censor self-identifying antibodies. Those that pass the censoring stage are used as imperfect detectors with a r-contiguous bits matching rule. Using this rule, string X and string Y are said to match if they agree in at least r-contiguous locations:

$$X: ABA\underline{DCB}AB$$

$$Y: CAG\underline{DCB}BA$$

In this example, X and Y match for $R \leq 3$. (FAPC94). These algorithms have been applied by Forrest's team to computer security (HF99), virus detection (FAPC94), and UNIX process monitoring (FHSL96).

The largest problem with this approach is that it is computationally expensive to generate all the required detectors. The analysis of the computational complexity is provided in (FAPC94). The number of required detectors is largely a function of the probability that two randomly generated strings match in at least r contiguous locations. The approximation holds only if $M^{-r} << 1$, which is often the case.

$$M = number \ of \ alphabet \ symbols$$

29

$$L = length \ of \ a \ string$$

$$R = number \ of \ contiguous \ matches \ required$$

$$P_m \approx M^{-r} \frac{(L-R)(M-L)}{M+1}$$

This probability leads to the number of initial detector strings that must be generated before censoring (N_{R_0}). The size of N_{R_0} is a function of the size of self (N_S), the reliability of the detectors (proability of a false negative detection, P_f), and the probability of a match (P_M).

$$N_{R_0} = \frac{-\ln(P_f)}{P_M \times (1-P_M)^{N_S}}$$

Although the number of initial detectors grows exponentially with the size of self, Forrest points out that this complexity affords a natural defense against spoofing. It is nearly impossible to change self and also change the detector to match (FAPC94). Another observation is that the probability of detection increases exponentially with the number of independently running detection algorithms (N_t).

$$P_{System \ fails \ to \ detect} = (P_f)^{N_t}$$

Therefore, the same system protection can be achieved through a distributed set of detectors, each with a smaller number of antibodies.

Later research uncovered two alternative algorithms for generating detector strings that execute in linear time (DFH96). These approaches essentially trade execution time for space. Both alternatives utilize data structures that grow exponentially in R, the number of contiguous bits in the matching rule. This paper also points out that these algorithms need to be fielded in a specific detection scheme. Several are suggested, but one of them, which mimics the organization of the BIS, suggests splitting up the antibodies between a number of autonomous agents, working in parallel. Another method enhances this approach with distributed, independent detection. Here, each agent's antibody set is generated independently. The biological analogy is a population of individuals, each with

slightly differing immune systems. The advantage with this architecture is that holes in one individual's detector set are covered by another's so that viral attacks cannot spread across entire networks. No method of cross-vaccination to protect individual computers containing holes is mentioned.

2.3.2 IBM AntiVirus.   Jeffry Kephart and his team at the IBM Thomas J. Watson Research Center have proposed a completely different immune system architecture, which integrates with IBM's AntiVirus product (KSSW97). The goal of this project is not a completely new way of fighting viruses based on the human immune system, but rather as a methodology to overcome the burden of detecting unknown viruses and generating scan strings from them (Kep94). This system uses a much looser abstraction of the biological model than that proposed by the UNM team.

The IBM immune system relies upon anti-virus heuristics and integrity monitors to detect the arrival of a yet unknown virus. It then uses carefully placed decoy programs to lure the possible new virus into infecting them. These infected decoys are then stored so that they cannot be executed. The decoys that have ingested viruses are analogous to macrophages that present viral fragments to T cell classifiers via the MHC (Kep94). However, the IBM system uses the infected decoys to automatically extract virus signatures strings for addition to the scanner database. The new signature string is chosen with respect to 500MB of known "self," or corpus, in order to ensure a low false positive rate (KA94). When a successful string is found, it is added to the local scanner database and proliferated to other users via database updates.

An automated proliferation mechanism has also been proposed that captures the clonal selection theory process of replicating known successful antibodies. This occurs via a kill signal and an accompanying vaccine (Figure 10). When an infection and subsequent viral signature extraction occurs, the neighbors of the host are notified and sent the new signature. If they are infected, the neighbors also in-turn propagate the new signature (Kep94).

This system only weakly integrates the biological model into a CVIS. However, it does integrate automation into the current IBM AntiVirus methodologies, thereby extending

31

- ○ Susceptible
- ● Infected
- ◉ Immune + Kill signal
- ⊛ Immune
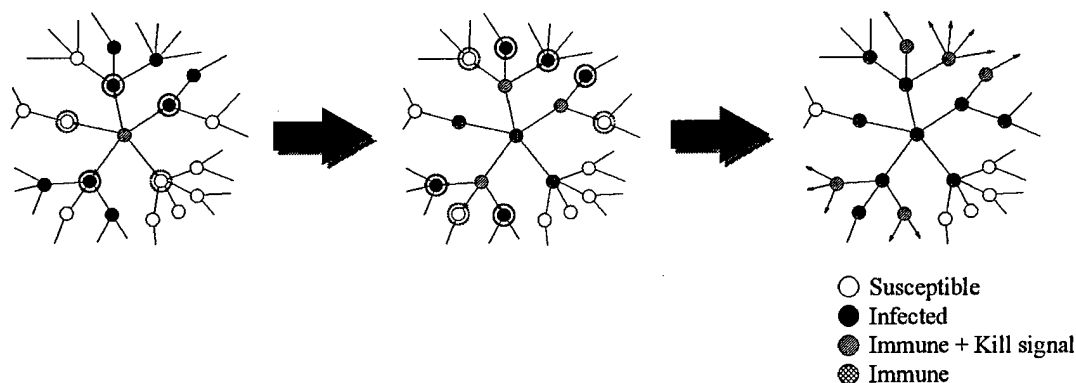
Figure 10. Kill signal propagation (Kep94).

their usability. However, the system is still limited by its need to maintain the virus scanner and the ever increasing scan string library. This represents an improvement, but not a revolution in current anti-virus capabilities. The other inherent flaw in this approach is the reliance on decoys for the capture of unknown viruses. With these sacrificial lambs, there is no guarantee that the virus will take the bait, which may result in a high false negative rate. These concerns are not addressed in Kephart's research.

2.3.3 Self-Adaptive Immune System. A third approach combines the negative selection algorithm, a genetic algorithm (GA), and the use of decoys to produce a self-adaptive CVIS (LMV99). Lamont, Marmelstein, and Van Veldhuizen point out the combinatoric problems with the other approaches and propose a multi-level, distributed architecture to relieve the high computational burden and hopefully afford real-time performance (Figure 11). The architecture utilizes many different intelligent, autonomous agents to provide sensing, recognition, and response mechanisms for the system.

The system of agents are managed in a 3-tiered architecture based on global, network, and local levels with each tier communicating though message passing. The agents at the local level contain all the functions necessary for tactical virus protection including detection, simple classification, elimination, and repair. This level relies upon decoys for capturing new, unknown viruses. The main purpose of the network level is to keep local AV agents aware of any other viruses found on the network. The network level attempts to

32

Global Level
— Resource Adaptation
— Detector Generation
— Resource Warehouse

Network Level
— Virus Classification
— Alert Generation
— Metrics Reporting

Messages
— Resource

Messages
— Resource Request
— Infected Decoy
— Metrics

Local Level
— Vulnerability Analysis
— Virus Detection
— Virus Elimination
— System Repair

Messages
— Resource
— Virus Alert

Messages
— Resource Request
— Infected File

Figure 11.   A hierarchical architecture for a self-adaptive CVIS (LMV99).

maximize antibody effectiveness within the limited memory of the system through the use of a GA. Finally, the network level also provides a conduit for information flow between the local and global levels. The focus of the global level is to generate adaptive AV resources and to evaluate system status. At this level, infected decoys are analyzed to extract signatures, which are sent down to the lower levels for fielding. The global controller also acts as the system warehouse for the storage of AV resources.

The hierarchical system design follows the patterns used in many organizations. This structure provides a framework for distributed decision making with local processing to improve efficiency and effectiveness. The system allows for fluid, redundant membership in the organization to facilitate ease of modification or expansion. The redundancy eliminates single points of failure within the system services and the use of distributed agents allows for parallel processing to eliminate bottlenecks.

This research proposes the use of a GA in order to perform affinity maturation on the decoys. The GA is used to evolve specific and generic decoy programs. The fitness of each decoy is evaluated as the number of files infected by the virus versus the processor duty cycle used by the decoy. The population of decoys subject to the GA is managed at the network level. Calculating the fitness function is somewhat straightforward, but the

33

problem lies in mapping the chromosome encoding of parameters such as the file name and directory locations. These fields have a large number of invalid combinations which must be managed. Also, the GA must maintain diversity within the population to ensure that yet unknown viruses are discovered. This approach also inherits all of the problems associated with using decoys, including a non-deterministic infection rate and the limited scope of discovering only file infectors (one of the least prevalent viral types (Wel98)). Finally, the advantage of the hierarchical approach is that infection can be stopped at the lowest level and those processes which require large computational resources can be moved to dedicated machines higher up in the tree.

2.3.4 Computer Health System. An alternative computational immune system architecture is based on bureaucracy, not biology (CO99). The proposed Computer Health System (CHS) is an informal and explanatory model based on the public health system. This model represents an approach to the anti-virus processes of detection, prevention, and cure of computer system infections based on a social enterprise. It is essentially a specific instantiation of a hierarchical management structure, very similar to the system proposed by (LMV99). This structure is envisioned to provide a framework for global computer virus protection, while immunological computation provides the local virus detection and elimination resident on individual computers.

The main goal of the CHS is to provide a framework for the protection of computer systems, with an emphasis on preventative strategies. These are largely implemented through information sharing. This requires network connectivity and secure communication channels in order to effectively, efficiently, and confidentiality disseminate computer medical information. This prevention is implemented by three levels, as suggested by the public health system:

- Primary – Prevention of the actual disease or injury, by reducing the exposure or risk level factors.

- Secondary – Identification and control of disease processes before symptoms are apparent.

- Tertiary – Prevention of disability by restoring individuals to their optimal level of function following some kind of damage.

The public health system has shown that prevention approaches that utilize the primary level of intervention have greater benefit for overall system protection. Four components are assigned the responsibility for implementing the CHS functions to accomplish the prevention levels (Figure 12). These specific areas can be mapped into the global, network, and local levels in the self-adaptive CVIS model (LMV99) however, the emphasis here is on the sharing of information to provide early prevention.

Specialized Agencies These are organizations or research groups, linked by common goals and objectives, that conduct research and trend analyses of viruses; develop useful statistics and metrics; perform general classifications for viruses; formalize the methods of virus detection, extraction, and repair; provide policy guidelines and standards; and assign responsibilities within the CHS.

Virus Experts This component provides a means of pooling resources and expertise. The virus experts are allocated the responsibilities for implementing and teaching preventative techniques, analyzing new viral types, extracting and diagnosing new viruses, and applying cures to the system after an infection.

Infrastructure This component provides a communications backbone that supplies system security, information sharing, and component interfacing to all the other components in the model.

Individual Systems Each computer within the CHS is equipped with a computer immune system based on the human immune system. The functions allocated to this component are system analysis, virus detection, self-adaptation, memory, and virus elimination.

The proposed CHS is a social structure, which is built to support a CVIS present on individual machines. Many of these structures currently exist, such as the virus experts and their newsgroup *alt.comp.virus*, but they are not organized into a global anti-virus team as proposed by this model. In order to organize the current structure into an effective team, this research proposes several, probably government, organizations including the
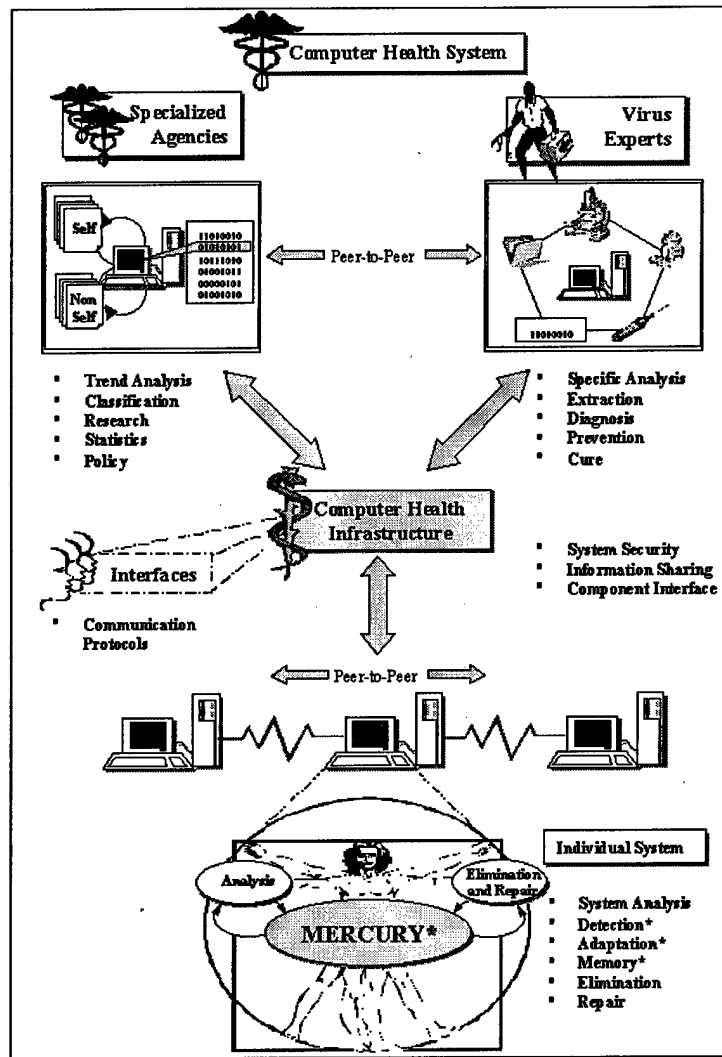
Figure 12.   Computer Health System architecture (CO99).

Virus Prevention Agency (VPA) and the Center for Virus Control (CVC). The VPA is an agency based upon the Federal Emergency Management Agency with similar anti-viral emergency management responsibilities, while the CVC accounts for the roles of the Center for Disease Control and Prevention in computer health. Cardinale and O'Donnell do not elaborate on any implementation details of the model or how current anti-virus organizations and structures would be mapped into it. Interestingly, it appears that many of the structures proposed have already self-organized including the virus experts and the specialized agencies. These can be found today in the laboratories of the commercial anti-

virus vendors and the computer virus forums, such as *Virus Bulletin*. The pieces missing are a protocol for the transfer of virus information via the infrastructure component and a CVIS for the individual systems. Once these are available, a specialized agency is all that is required to make the CHS a reality. Unfortunately, these missing pieces are not trivial, technically or bureaucratically.

As a partial answer to the current technical shortcomings of the CHS, (CO99) also propose a machine learning approach for the generation of antibody search strings, based on constructive induction. The prototype system, MERCURY, is a virus detection component made up of a virus scanner, a constructive induction based learning engine, and a knowledge base. The induction engine, HEC, provides the scanner with byte signatures that distinguish between self and non-self. MERCURY is designed to provide the human immune system functions of detection, adaptation, and memory to a CVIS. MERCURY would have to further interface to a system that provided analysis and virus elimination/repair in order to fully encompass the features allocated to the CHS individual systems component.

The constructive induction engine generates, orders, evaluates, and incorporates hypotheses. This process outputs a set of detectors, based on the hypotheses that best define the rules of self and non-self. These detectors are then used by the virus scanner to check for viral infections within the file system.

This research was not able to validate, or refute, that constructive induction provides a suitable learning mechanism for the generation of virus detectors. The experimentation did show an exponential computational complexity in the generation of hypotheses (Figure 13). The long execution times would render MERCURY inapplicable to an unobtrusive, real-time implementation without algorithmic improvements. But, although it was not efficient, this approach was effective at detecting self and non-self files within varying degrees of accuracy.

2.3.5 Immunity for Machine Learning. Dr's John Hunt and Denise Cooke at the University of Wales, Aberystwyth have looked at the machine learning aspects of artificial immune systems and applied them to data mining with their ISYS project (Her96). The system has been used for fraud detection in mortgage loan applications and consumer pat-
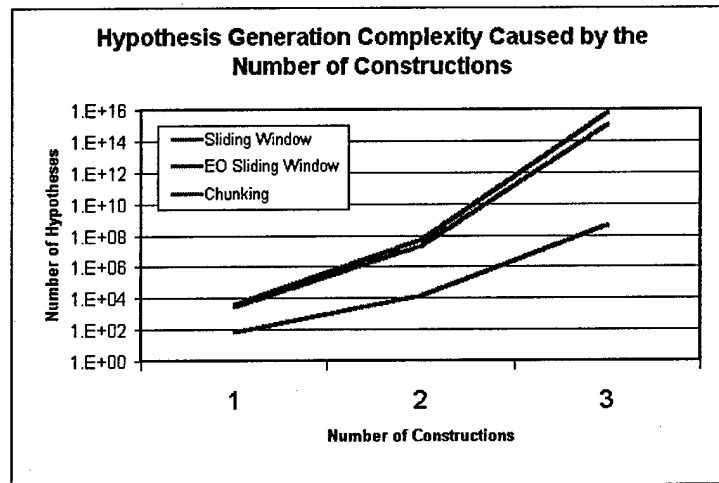
Figure 13. HEC hypothesis generation complexity (CO99).

tern extraction from supermarket sales data. The later is then used for targeted marketing campaigns.

The ISYS system can be used in two ways. ISYS utilizes case-based reasoning that relies on a training set. After which, the cells can be examined for common patterns. This is analogous to the biological primary response. The other use is to apply the trained system against a new data set in order to recognize previously seen patterns. This use is similar to a secondary response (HC96).

ISYS has been shown to be successful at data mining, especially over large data sets. Furthermore, it has outperformed neural networks in terms of learning speed (Her96). These successes, combined with the ability to forget unused antibody sequences, gives promise to a computer virus artificial immune system.

The ISYS model emphasizes pattern matching and learning through the use of hypermutation and affinity maturation. A root object is surrounded by a network of detector cells. These detectors produce an antibody string via an algorithm that mirrors the combinatorics of the biological genetic process. The detector implements imperfect matching with a threshold. When a match exceeds the threshold, the cells multiply with possibly mutated clones. These are then added to the network (HC96). Further details are not given, but presumably the cells which are not activated are deleted from the network in

Table 3. Summary of artificial immune system features.

	UNM	IBM	Self-Adaptive	CHS	ISYS
Emphasis	Negative Selection	Use of Current Systems	GA, Distributed Architecture	Prevention, Human Agencies	Data Mining
Antibody Creation	Random	Capture	GA	Constructive Induction	Case-based Reasoning
Matching Method	R-bits	Perfect Match	Not Specified	Perfect Match	String Permutation
Detectors	Strings	Decoys, Strings	Decoys, Strings	Strings	Strings
Clonal Selection		√	√	√	√
Affinity Maturation			√		√
Complete System		√	√	√	√
Distributed	Suggested	√	√	√	

order to minimize resource utilization. The reason or utility for creating a network, as opposed to just a list of detectors is also not given.

2.3.6 AIS Summary. The biological model has been interpreted differently by many researchers in its transformation to the information system domain. The computer health system even suggests a model of operation based on governmental organizations. Due to these variations, each one has a different emphasis and offers a range of capabilities. A summary of the similarities and differences can bee seen in Table 3.

2.3.7 Agent-based Immunity. The use of agents as an architecture for implementing an artificial immune system is suggested by Dasgupta (Das98). The highly parallel and distributed notion of the biological immune system implies that an integrated architecture can be viewed as a multiagent system, where separate functions are carried out by individual agents (Das98). Furthermore, the general immune system features represent a model of adaptive processes at the local level, with useful behavior emerging at the global

level (Das99). This is similar to the description of multiagent system (MAS) operations by the artificial intelligence community (DeL99a). In Dasgupta's architecture, agents not only communicate and collaborate, but also migrate in order to further monitor and analyze the environment. No implementation details nor the performance issues of using mobile agents are addressed, but the arguments for the use of agents are compelling.

2.4 Agents

Despite all the research into software agents, researchers do not seem to agree on what exactly an agent is (FG96). One definition speaks of agents as software components that communicate with their peers by exchanging messages (GK95). This paper focuses on inter-agent communication. Another paper on mobile agents uses the definition of a software agent as a program that can halt itself, ship itself to another computer on the network, and continue execution at the new host (Ven97). The definition of an agent changes from paper to paper. Therefore it appears that the definition of an agent needs to be agreed upon up front in a discussion in order to communicate effectively therein. The broadest definition includes anything that can be viewed as perceiving its environment and acting upon that environment (RN95). Because the idea of what exactly constitutes an agent is controversial, it has been suggested by Russel and Norvig that the idea of an agent be used as a tool for analyzing systems, and not as an absolute delimiter between a world of agents and non-agents (RN95). So, to avoid absolute classification, agents are also typically described as possessing one or more of the following characteristics (Sun98), which can be used to further classify them in useful ways (FG96):

- Autonomous

- Adaptive/Learning

- Mobile

- Persistent

- Goal oriented

- Communicative/Collaborative

40

- Flexible

- Active/Proactive

Most of the current research is related to embodying agents with these qualities. But, to solve complex problems, groups of agents must work together, often in heterogeneous environments (DeL99a).

2.4.1 Multiagent Systems. Research into multiagent systems (MASs) is concerned with the study, behavior, and construction of a group of agents that interact with each other. Currently, the majority of agent-based research involves only a single agent, but the need for more complex systems that contain multiple agents that communicate has become a reality (Syc98). The rise of inter-networking, distributed computing, and the Internet has only fueled the need for more complex agent-based systems. A MAS is defined as a loosely coupled network of problem solvers that interact to solve problems that are beyond the scope of each individual agent's capabilities or knowledge (Syc98). In these MASs, individual agents follow their own goals and what emerges is a system level behavior (DeL99a).

The use of multiple agents in a system design can bring to the implementation many advantages. Sycara has identified six key areas where the use of MASs has applicability or reveals certain advantages (Syc98):

1. To solve problems which are too large for a single, centralized agent to solve due to resource limitations, performance bottlenecks, or an inherent single point of failure.

2. To allow for the communication and interoperability between multiple existing legacy systems.

3. To solve problems that can be regarded as a population of interacting agents.

4. To provide solutions that efficiently utilize distributed information sources.

5. To provide solutions to problems where expertise is distributed.

6. To enhance system performance in the areas of computational efficiency, reliability, extensibility, maintainability, responsiveness, flexibility, and reuse.

41

The MAS advantages in the areas indicated are somewhat overshadowed by the challenges in MAS design. These challenges are the result of the complex interactions that can occur in agent populations. Sycara identifies six challenges to MAS design (Syc98):

1. How to decompose and allocate problems among multiple agents and then synthesis the results?

2. How to enable agents to communicate and therefore interact?

3. How to ensure that agents act coherently to avoid unstable system behavior?

4. How do individual agents reason about the actions of others and coordinate appropriately?

5. How are conflicts and differing viewpoints between agents resolved?

6. How are MASs engineered and constrained?

These challenges must be addressed during the system design process in order to create and effective and efficient multiagent system.

2.4.2   Mobile Agents.   Adding mobility to the agent paradigm gives the agent the ability to migrate autonomously from node to node on a network (KT98). Users can send mobile agents on a journey to roam the network on a predefined path, or a dynamic one based on the agent's goals. After accomplishing its defined tasks remotely, the agent can return to the source node to report the results (KT98). Mobile agents are defined somewhat more formally as objects that have behavior, state, and location (Som97).

Mobile agent-based computing can be viewed as an extension of remote script execution or the remote submission of batch jobs (HCK95). Because of this legacy and the desire to operate in heterogeneous environments, mobile agents are typically written in a scripting or interpreted language. This induces a performance penalty versus the use of native code, but besides heterogeneous platform execution, it can also give advantages in the area of security (HCK95). This security advantage can be gained if the interpretation environment can rigorously control agent access to system resources, something the Java sandbox security model has not entirely been successful at (Lad97).

42

The use of mobile agents promises to reduce network use, increase synchrony between clients and servers, add client-specific functionality to servers, and introduce parallelism in program execution (KT98). Only general statements of these types are made throughout the literature (KT98, HCK95, Syc98, Mah00), with few applications that reap the benefits from this paradigm. Karnik and Tripathi state that little has been done in the area of quantifying mobile-agent performance tradeoffs and that at this time they believe that mobile-agent systems have yet to reach maturity (KT98). This is somewhat mirrored by the IBM study, which concludes that there are many individual areas where mobile agents have advantages, but none are overwhelming and that in almost every case, an equivalent solution can be realized through other methods, such as remote procedure calls or message passing (HCK95). The aggregate advantages of mobile agents as pointed out in the IBM whitepaper essentially boil down to the creation of a framework for personalized network services. This advantage is overwhelmed by disadvantages in the areas of security and transmission efficiency.

The major obstacle preventing the usage of mobile agents is security (KT98). The ability to add client-specific functionality to servers immediately conjures up the idea of mobile agent viruses. This requires the use of a virus immune system and a trust architecture based on authentication and encryption (HCK95). The mobile agent security issues remain an open research area as no current systems completely address the security problems (KT98).

The performance issues of packaging and sending an agent, its methods, attributes, and current state versus passing only data using simple datagrams have not been addressed or quantified (KT98). Mobile agents would only gain a network performance advantage when the amount of data sent via messages exceeds the agent code itself. This may only be true in the case of remote file system access or large database queries. In many other applications, the load balancing decision between remote and local agent execution may not be known until that point is exceeded, especially in irregular problems, such as searching.

2.4.3 Agent Software Engineering. Agent-based systems are among the most complex to construct because the autonomy in the individual agent behaviors inherently

contributes to the overall system complexity (HJTW99). Agent-based software engineering was created to facilitate the design of software able to operate in an environment where programs are written by different people, in different languages, and with different interfaces (GK95).

    2.4.3.1 Multiagent Systems Engineering. Multiagent Systems Engineering (MaSE) is a software engineering approach for multiagent systems proposed in (DeL99a). Much of the work in the agent field has focused around the artificial intelligence aspects as well as the makeup of individual agents themselves. Very little has been done in the area of MAS design and construction. When systems of multiple agents are constructed many problems arise (Section 2.4.1), as identified by Sycara (Syc98). DeLoach attempts to answer Sycara's sixth challenge through this research, but interestingly, problems 1 and 2 are also addressed through the MaSE process of system decomposition.

    MaSE is an abstraction above the object-oriented (OO) paradigm where agents occupy a level above traditional objects. In so doing, the author uses a broad definition of agent, which may encompass whatever aspects of agenthood the user of MaSE wishes to implement. By modeling agents as "active objects," MaSE extends the typical object model of passive methods and attributes, to include goals and a common communication language. To do so, the common object modeling techniques of Rumbaugh and the Unified Modeling Language (UML) are extended to include the semantics required to model agency, cooperative behavior, and communications. The sum total of these semantics are known as AgML, or Agent Modeling Language. Therefore, MaSE is an extension to UML. MaSE not only can be seen as a proposal for the methodology, but also as a how-to guide for using it to design MASs.

    The MaSE methodology is a four step process (Figure 14). One unique aspect of this is that the general components of the system are designed before the system itself. This is consistent with a MAS design as the system is made up of a population of interacting entities that may in fact be dynamic. The domain level design is concerned with the agent types, goals, and external interfaces. This is analogous to breaking a problem statement into objects in the traditional object-oriented analysis (OOA) methodology. The agent

level design defines the components within each agent and how they interact. The component design completes the low level design of each agent's components. Finally, system design arranges the number and types of agents into an overall system framework of collaborators. To accomplish these steps, AgML is used to graphically diagram each agent, their communication, and the system architecture.
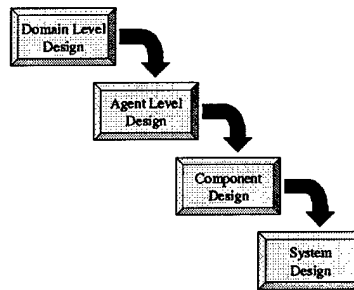
Figure 14.   MaSE methodology (DeL99a).

MaSE represents a new software engineering methodology for designing MASs. Because of this, no tools to automate this process have been developed as exist for UML, such as Rational Rose. Additionally, the methodology is unproven in actual use. Despite a lack of tools and general acceptance, the methodology appears useful for MAS design as it could provide the rigor and visualization needed for complex multiagent software engineering tasks.

2.4.3.2 Agent Design Patterns.   Instead of a design process, Aridor and Lange propose design reuse for agent application engineering (AL98). They advocate the use of agent design patterns as a way to improve the agent design process and make agent applications more flexible, understandable, and reusable. The idea of design patterns originated in the OO community and has been recognized as one of the key innovations in that field (AL98). Since agents have grown out of object-oriented design (OOD), it only makes sense to also migrate the design patterns to agent architectures. The focus of this paper is on the initiation of agent design patterns for mobile agent applications. Although targeted for mobile agents, the authors point out that the pattern idea can be applied to non-mobile agent systems with equal benefit.

45

The patterns proposed are decomposed into traveling, task, and interaction types. Of these, only the traveling patterns are mobile agent specific. The master-slave and plan patterns of the task type and the meeting forms of the interaction type could all be realized with distributed, non-mobile agents. By utilizing the master-slave pattern as a base, (AL98) develops a file searcher agent that searches for files in a remote file archive. This has a direct application to an anti-virus scanner and also a mobile agent virus itself. Aridor and Lange ignore the entire issue of security with these patterns.

In addition to the issues of security and assurance, very little is said about system performance. The paper claims that mobile agents reduce network traffic and provide an effective means of overcoming network latency. This can only occur when the cost of message passing is greater than the cost of marshaling an agent, its methods and attributes, and sending the entire package down the wire. This fact is only alluded to in the applicability of interaction agent patterns and nowhere is a dynamic movement decision capability included in the patterns.

Although admittedly very pioneering in the agent design field, the patterns proposed in this paper contain very little substance as to make them largely unusable other than to bring attention to the goal of constructing a design pattern library. In their current state, the patterns offer little to overcome the issues of mobile agents or toward answering the issues in multiagent system design, such as those put forth by Sycara (Syc98).

2.4.4 Agent Development Libraries. Along with the theoretical work on agents, several libraries, frameworks, or agent development kits (ADKs) have been developed. A list of numerous academic and commercial ADKs can be found at the Agent Builder commercial site (Ret99). Many of the initial ADKs were developed in scripting languages, such as TCL or Telescript, to facilitate operation in heterogeneous environments. But, with its network-centricity, sandbox security model, and platform independence, Java has become the environment of choice for the development of agent-based tools (Som97). The problem with these interpreted languages is speed, and very little research has been done on the effects of this on high-performance computing applications.

2.5 Summary

This chapter discusses background information and current research within several of the system integration domains. Included is background on computer viruses, the human immune system, and software agents. Additionally, current research related to this effort is reviewed in the areas of artificial immune systems and agents. These topics and ideas are integrated into the beginnings of the system design of an agent-based computer virus immune system in the next chapter.

III. Computer Virus Immune System Design Elements

This chapter discusses several elements that are prerequisites to system design. These areas include the design methodology, problem domain definition, matching rule selection, immune system model creation, and the implementation language selection. The selection of each of these imposes certain restrictions on the actual system design and implementation. The chapter begins with a description of the system design methodology.

3.1 Design Methodology

A design methodology should provide a framework for defining the requirements, architecture, and implementation of a complete and fully operationally multiagent CVIS. This methodology is a process that closely follows the software engineering sequential model steps of analyze, design, code, and test (Pre97). However, this design also requires the integration of agents and their unique requirements. Therefore, the Multiagent Systems Engineering (MaSE) process has been added into the design phase to facilitate the integration of agent characteristics into the system objects (Figure 15).
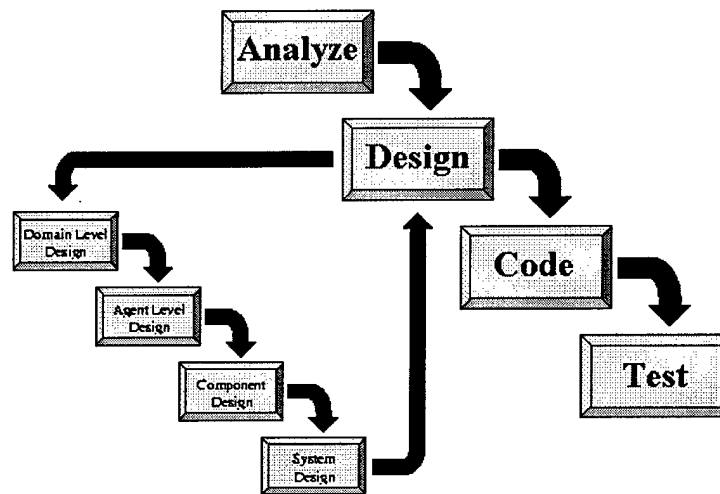
Figure 15.   Design process.

The MaSE methodology is a four step process. The domain level design is concerned with the agent types, goals, and external interfaces. This is analogous to decomposing a problem statement into objects in the traditional object-oriented analysis methodology.

48

The agent level design defines the components within each agent and how they interact. The component design completes the low level design of each agent's components. Finally, system design arranges the number and types of agents into an overall system framework of collaborators. To document these steps, an extension of the unified modeling language (UML), the agent modeling language (AgML) is used. AgML captures the agent design by graphically diagramming each agent, their communication, and the system architecture. This multi-step software engineering process begins with the analysis phase to understand the problem domain and the biological model being applied to it. (Figure 16).

Analyze · **Understand Problem Domain**
· **Create Biological System Model**
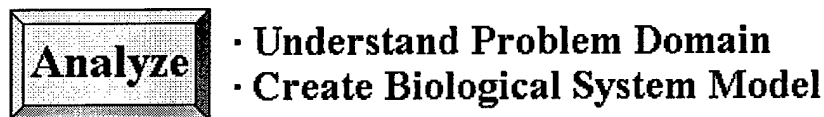
Figure 16.   Analysis milestones.

Next, the MaSE methodology is followed until a final system design and deployment is created (Figure 17). The design phase begins with the initiation of the MaSE domain level design through the development of use-cases. The component level design includes a trade study comparing the available detector string matching rules. The best algorithm is integrated into antibody application operations.

Design · **Domain Level Design**
· **Agent Level Design**
· **Component Design**
· **System Design**
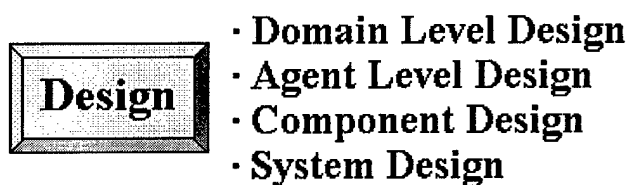
Figure 17.   Design milestones.

The coding phase not only includes the actual implementation of the system, but also an evaluation of the most appropriate implementation environment for the multiagent CVIS (Figure 18).

The last step is the development and execution of the test plan. As part of this test plan, system performance metrics need to be developed in order to understand the
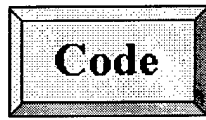
Figure 18.   Coding milestones.

efficiency, effectiveness, and scaleability of the system (Figure 19). But, well before testing can begin, the problem domain must be analyzed and understood.
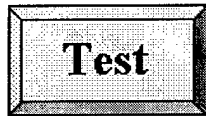
Figure 19.   Testing milestones.

3.2   Problem Domain

The major objective of this prototype system is to detect the existence of non-self patterns within a potentially larger set of existing self patterns. The problem domain is over the set X of finite length symbol sequences. X is typically represented as $X \in \{0,1\}^l$, or $X \in \{0 \ldots 255\}^{\frac{l}{8}}$, but the exact representation is an implementation detail. Set X contains two subsets, self, $S \subseteq X$, and non-self, $N \subset X$ such that $S \cup N = X$ and $S \cap N = \emptyset$. The non-self patterns represent malicious, viral code, while the self set is indicative of legitimate, benign programs (Section 2.1).

The task of the detection algorithm is the classification of an input pattern, $I \in X$, as either self, or non-self; as either benign or malicious. Given a binary representation, an input string I, matching threshold ϵ, a detector set D, and a matching function f, $match(f, \epsilon, I, D) \rightarrow \{malicious, benign\}$. The antibody detection strings, α, are assumed to be certified as non-self patterns via the negative selection algorithm (Sections 2.2.3.1, 2.3.1, and Figure 9).

$$I \in \{0,1\}^l$$

$$\alpha \in \{0,1\}^k, \qquad k \leq l$$

$$D = \{\alpha_1, \alpha_2, \ldots, \alpha_i\}, \qquad i \in \aleph$$

$$f(I,\alpha) \to \{p : \Re \mid p \geq 0 \wedge p \leq 1\}$$

$$match(f, \epsilon, I, D) = \begin{cases} malicious : & f(I,\alpha) \geq 1 - \epsilon \\ benign : & otherwise \end{cases}$$

This detection methodology can generate two types of errors, Type I, or false positive errors and Type II, or false negative errors. A false positive error, δ^+, occurs when a member of the self set, S, is incorrectly classified as malicious. Conversely, a false negative, δ^-, is the classification of a member of the non-self set, N, as benign.

$$(I \in S \cap match(f, \epsilon, I, D) = malicious) \to \delta^+$$

$$(I \in N \cap match(f, \epsilon, I, D) = benign) \to \delta^-$$

The core functionality of the detection process is provided by the matching rule function, f. The proper selection of a pattern matching function is instrumental in reducing the Type I and Type II errors.

3.3 Matching Rule Selection

The natural immune system implements two core functions, detection and elimination of pathogens (Section 2.2). This proposed computer virus immune system is no different. The crux of the problem is the detection of malicious code that has penetrated the boundaries of the system so that it cohabitates with a much larger self set. This is an application of pattern matching between a set of antibody scan strings and the set of data residing on the local node.

Pattern recognition is a process by which input data is discriminated, not between individual patterns, but between populations. This is accomplished through a search for features or attributes common to members of the various populations or sets (TG74). The biological immune system accomplishes this through the physical and chemical binding of antibodies to antigen molecules. Because of the negative selection process, a match

51

is a segregation of that molecule into the set of non-self. In the computational domain, this process is completed by string matching rules. Like the body, the goal is to utilize imperfect detectors to recognize non-self with a low false positive rate and a high probability of detection.

3.3.1 Matching Rules. The many pattern matching functions come in two varieties, distance measures that express how different two sequences are, and similarity functions which measure how alike they are (NS93). Intuitively, objects that are close together in the feature space must be similar, while those that are farther apart are dissimilar (NS93). Those that are similar to a non-self pattern within a certain threshold can be classified as non-self. The matching rules investigated in this study utilize statistical, physical, and binary distance measures of distance or similarity. One statistically based similarity measure is the correlation factor, or correlation coefficient.

3.3.1.1 Statistical. The correlation coefficient produces a number between -1 and 1 that relates how similar the two input sequences are. It is defined as:

$$X, Y \in \{0 \ldots 255\}^N, \qquad N = \frac{l}{8}$$

$$\rho = \frac{\sum_{i=1}^{N}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{N}(X_i - \overline{X})^2 \sum_{i=1}^{N}(Y_i - \overline{Y})^2}}$$

The most common implementation of this measure is ρ^2, which is somewhat easier to compute (NS93). Other common matching rules operate at the bit level.

3.3.1.2 Binary Distance. The correlation coefficient utilizes the byte values of the input and antibody strings. However, at their lowest level these strings are sequences of bit values. Therefore, it makes sense to utilize difference and similarity measures that operate in the digital domain. The most obvious is the Hamming distance, which counts the number of bit features that are different between two strings. Taking the complement

52

results in the number of bit positions that are alike (NS93).

$$Hamming\ Similarity = \sum_{i=1}^{N} \overline{(X_i \oplus Y_i)}, \qquad X, Y \in \{0,1\}^N$$

The Hamming distance is the "gold standard" for measuring the distance between bit strings, but to be more useful, several authors have proposed additional similarity measures that extend the Hamming distance to produce the relative number of features that match or differ (NS93). These matching functions utilize the following definitions:

$$X, Y \in \{0,1\}^N$$

$$a = \sum_{i=1}^{N} \zeta_i, \qquad \zeta_i = \begin{cases} 1: & X_i = Y_i = 1 \\ 0: & otherwise \end{cases}$$

$$b = \sum_{i=1}^{N} \xi_i, \qquad \xi_i = \begin{cases} 1: & X_i = 1, Y_i = 0 \\ 0: & otherwise \end{cases}$$

$$c = \sum_{i=1}^{N} \gamma_i, \qquad \gamma_i = \begin{cases} 1: & X_i = 0, Y_i = 1 \\ 0: & otherwise \end{cases}$$

$$d = \sum_{i=1}^{N} \psi_i, \qquad \psi_i = \begin{cases} 1: & X_i = Y_i = 0 \\ 0: & otherwise \end{cases}$$

These basic measures are combined into many different similarity functions with the goal of producing a better similarity coefficient.

Russel and Rao:

$$f = \frac{a}{a+b+c+d}$$

Jaccard and Needham:

$$f = \frac{a}{a+b+c}$$

Kulzinski: A one has been added to the denominator of the author's equation to avoid division by zero errors. Due to the definition of b and c, this will occur whenever there is an exact match.

$$f = \frac{a}{b+c+1}$$

Sokal and Michener:

$$f = \frac{a + d}{a + b + c + d}$$

Rogers and Tanimoto:

$$f = \frac{a + d}{a + d + 2(b + c)}$$

Yule:

$$f = \frac{ad - bc}{ad + bc}$$

3.3.1.3 Landscape Affinity Matching. The biological immune system "identifies" antigen by physically and chemically bonding with it. Only the correct inverse protein structure and chemical make-up will bind with a high enough affinity to attach to an antibody or MHC protein.

In most AISs, this binding is performed by bit or byte string comparisons (Kep94). Others extend bit matching to account for imperfect matches by using the Hamming distance, or r-contiguous bits (HF99, FAPC94). Another extension is to present combinatoric variations of the non-self string to the detector in order to extend the search space of a specific matching function (HC96, HF99). All of these variations capture the chemical and physical matching process at a fairly high conceptual abstraction. Along with the historical matching rules, this study also introduces three more, dubbed landscape affinity matching.

In this methodology, the input strings are sampled as bytes and converted into positive integer values in order to generate a skyline, or landscape. The antibody strings are similarly represented. The antibody and input landscapes are compared in a sliding window fashion (Figure 20).

The comparison can be made in several ways that produce and affinity measure. Those used are difference, slope, and physical affinity. These measurements are then checked against a threshold value. If the affinity exceeds the threshold, a match is declared (Figure 21). The input string and the antibody are sequences of bytes compared N at a time.

$$X, Y \in \{0..255\}^{N}$$

54
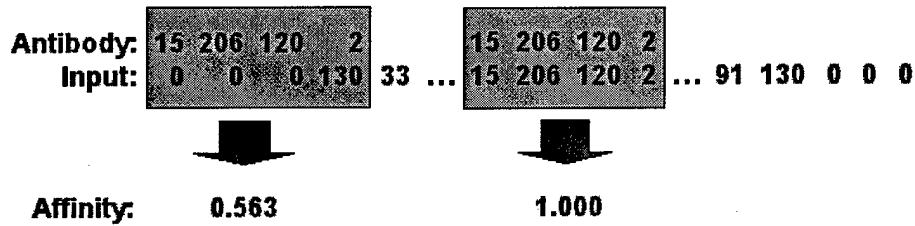
Figure 20. Landscape affinity matching representation and windowing.

In the difference matching rule, the differences in the string bytes are simply summed.

$$f_{difference} = \sum_{i=1}^{N} |(X_i - Y_i)|$$

The slope matching rule looks at the differences in the changes between bytes among the two stings.

$$f_{slope} = \sum_{i=1}^{N-1} |(X_{i+1} - X_i) - (Y_{i+1} - Y_i)|$$

Physical matching stacks the two strings, like a game of Tetris, and then calculates the resulting gaps between the two strings (Figure 21).

$$f_{physical} = \sum_{i=1}^{N} (X_i - Y_i) + 3 \times |\mu|, \qquad \mu = min(\forall i, (X_i - Y_i))$$
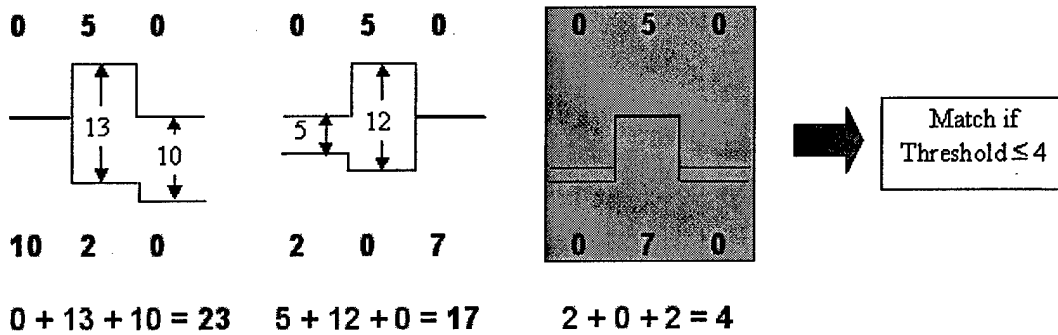
Figure 21. Physical landscape affinity matching methodology.

55

Landscape affinity matching captures the ideas of matching the biochemical, physical structure and imperfect matching with a threshold for activation. The differences between the input landscape and the antibody "heights" can be likened to the ease of chemical bonding between proteins. The closer the peaks and valleys are, the greater the likelihood of a bond and the higher the affinity.

3.3.2 Comparison Criteria. In order to compare these 12 selected matching rules, each one is calculated with a common data set. A random string of 32 bytes is generated as the input string. From this, 4 bytes are selected from positions 11-14 to act as an antibody string. Therefore a known exact match is always present at position 14. The 4 byte antibody is compared with the 0 padded input string in a sliding window fashion. This generates 35 measurements of difference or similarity for each matching rule.

All measurements are converted to similarity measurements and normalized so that a value of 1 represents an exact match, while a 0 is produced by the two most dissimilar strings. This test is run on 5 random input strings to produce a statistical sampling of the rules' performance. In order to compare the effectiveness of the various methods, an average signal to noise ratio (SNR) is calculated, along with a function value distribution.

3.3.3 Results and Analysis. The SNR is a measure of a matching rule's ability to accurately discriminate a match signal from all the non-matches (noise). It is calculate as 10 times the log of the ratio of the signal power to the average noise power. In order to equate with communications theory, in this application the normalized rule function values are interpreted as voltages driving a normalized resistor.

$$\frac{S}{N} = 20 \times \log(\frac{1}{\eta}), \qquad \eta = \frac{1}{N-1} \sum_{i=1}^{N-1} x_i, \qquad x_i \neq 1$$

The results can be seen in Figure 22. A large SNR indicates a more specific detector, while a low value is indicative of a general detector. A specific detector will be able to find a pathogen with a low false alarm rate. As the SNR decreases, the probability of

56

generating a false positive detection increases. However, a general detector is able to cover a larger subset of the self/non-self space.
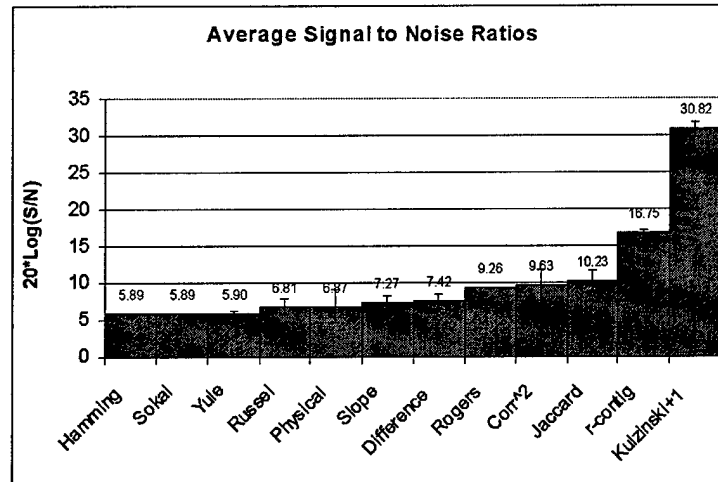
Figure 22.   Average signal to noise ratios.

The Kulzinski measure produced a disproportionately large SNR. This measure would produce the most specific detector. The Hamming distance and the Sokal functions produce the lowest signal to noise ratio, pulling a signal only about 6dB above the noise floor. These would result in much higher false alarm rates on average. Interestingly, the landscape affinity measures did not perform much better. The r-contiguous bits rule produced a SNR of almost 17. The increased stringency in this rule compared to the Hamming distance, where matching can occur anywhere, results in a detection rule that is almost 3 times more specific than the Hamming distance.

For this application, a balance between generality and specificity in the detector is desired, with a tendency towards the specific. A general detector allows the antibody to cover a greater portion of the non-self region, at the expense of possibly overlapping a small portion of self (false positive). For the CVIS, it is desirable to be able to increase the sensitivity of the detector by reducing the detection threshold. This allows the system to increase its awareness for a possible infection. For this reason, a matching rule with the ability to pull the signal out of the noise floor, but not too high is desirable. A signal to noise ratio between 9 and 12 is probably sufficient, which corresponds to the Rogers,

correlation coefficient squared, and the Jaccard measures. In order to down select among these, the function value distributions are plotted.

The various values produced by the comparison functions are scaled and plotted using histograms in order to understand the density functions of the various measures. These can be seen in Figures 24 and 25. Ideally, the density function for this application should approximate Figure 23. This corresponds to a signal to noise ratio of 8.05dB. The ideal density function would allow for a low false positive rate with a smooth scaling in sensitivity as the detection threshold is moved to the left. In the ideal case, the density function value at 90%-100% should be $\frac{1}{35} = 0.0286$, which indicates only one exact match and all other similarity values are less than 90%. Additionally, a low variability, especially in the higher affinity values is desired. This would indicate consistent performance from the detector.
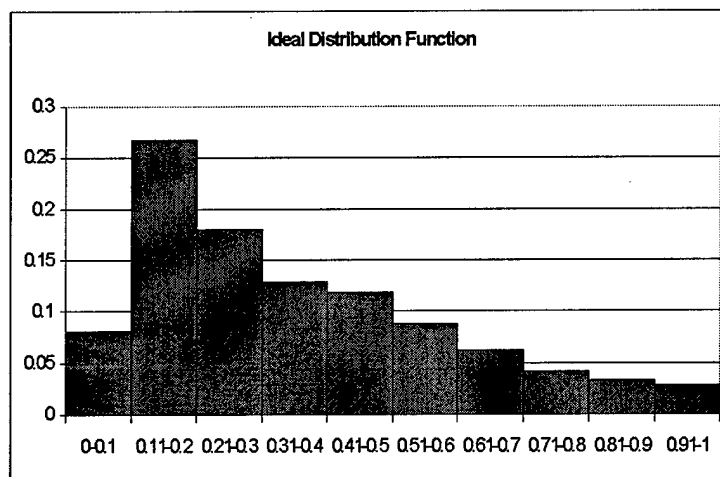
Figure 23.   Ideal matching rule distribution function.

Evident in these histograms are the reasons for some of the signal to noise ratio values as well as confirmation of the generality or specificity of the matching rules. The Kulzinski measure's histogram dramatically depicts this rule's ability to perform as a highly specific detector. Likewise, the r-contiguous bits is quite heavily weighted at the lower affinity end. The landscape affinity physical measure produces the worst discriminator, with a naturally high false positive frequency along with large variability in all the other value

58
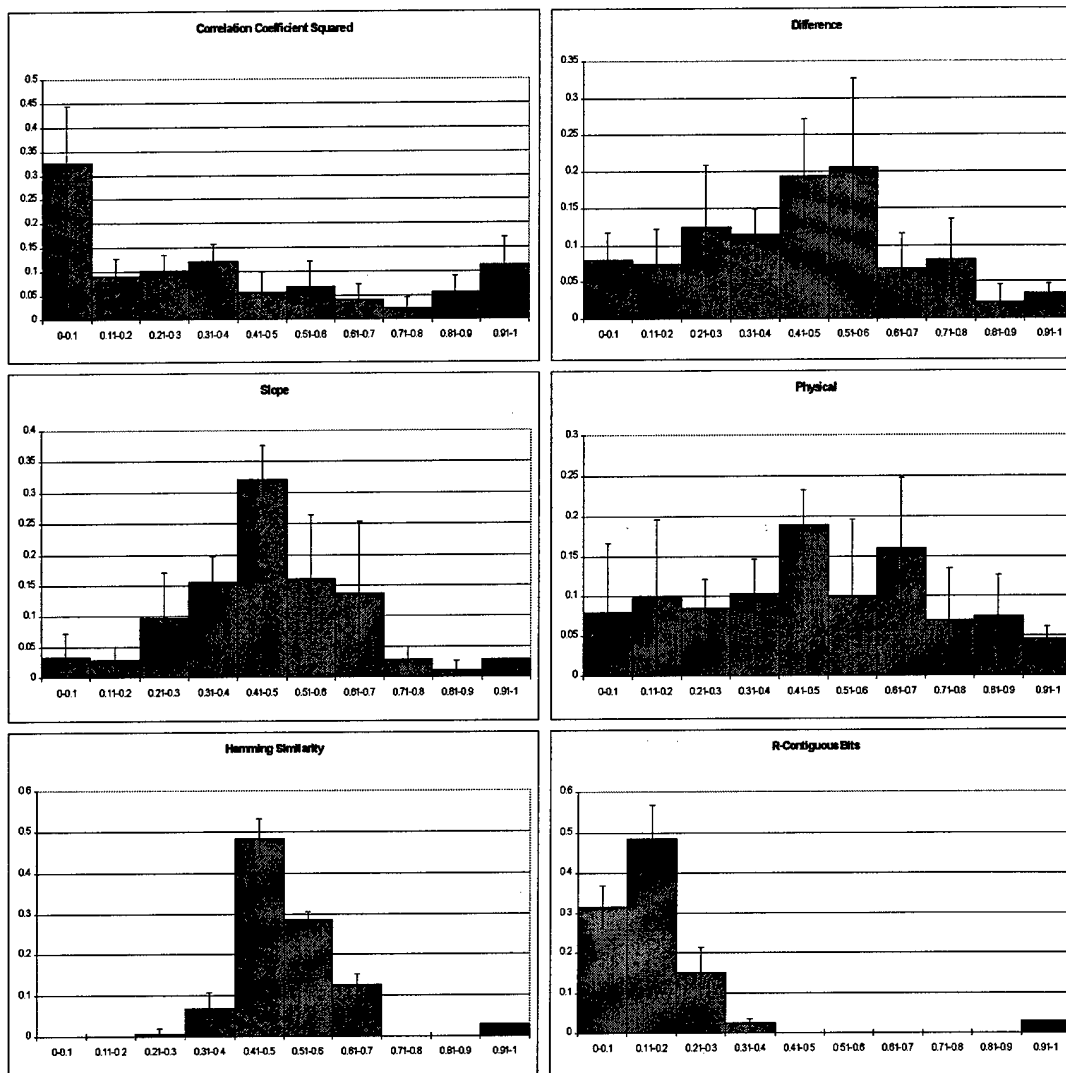
Figure 24. Normalized matching rule distribution functions part I.

bands. Additionally, it has an almost uniform distribution in value frequencies, indicating pour discrimination. The Hamming distance and Sokal's measure, which posses identical SNR values also show their equality in their density functions. This is because the Sokal and Michener function is equivalent to the normalized Hamming similarity.

Based on their signal to noise ratios, Rogers, the correlation coefficient, and the Jaccard measurements are the most applicable to this application. Both the Rogers and Jaccard rules produce distribution functions that are only slightly better than the Ham-
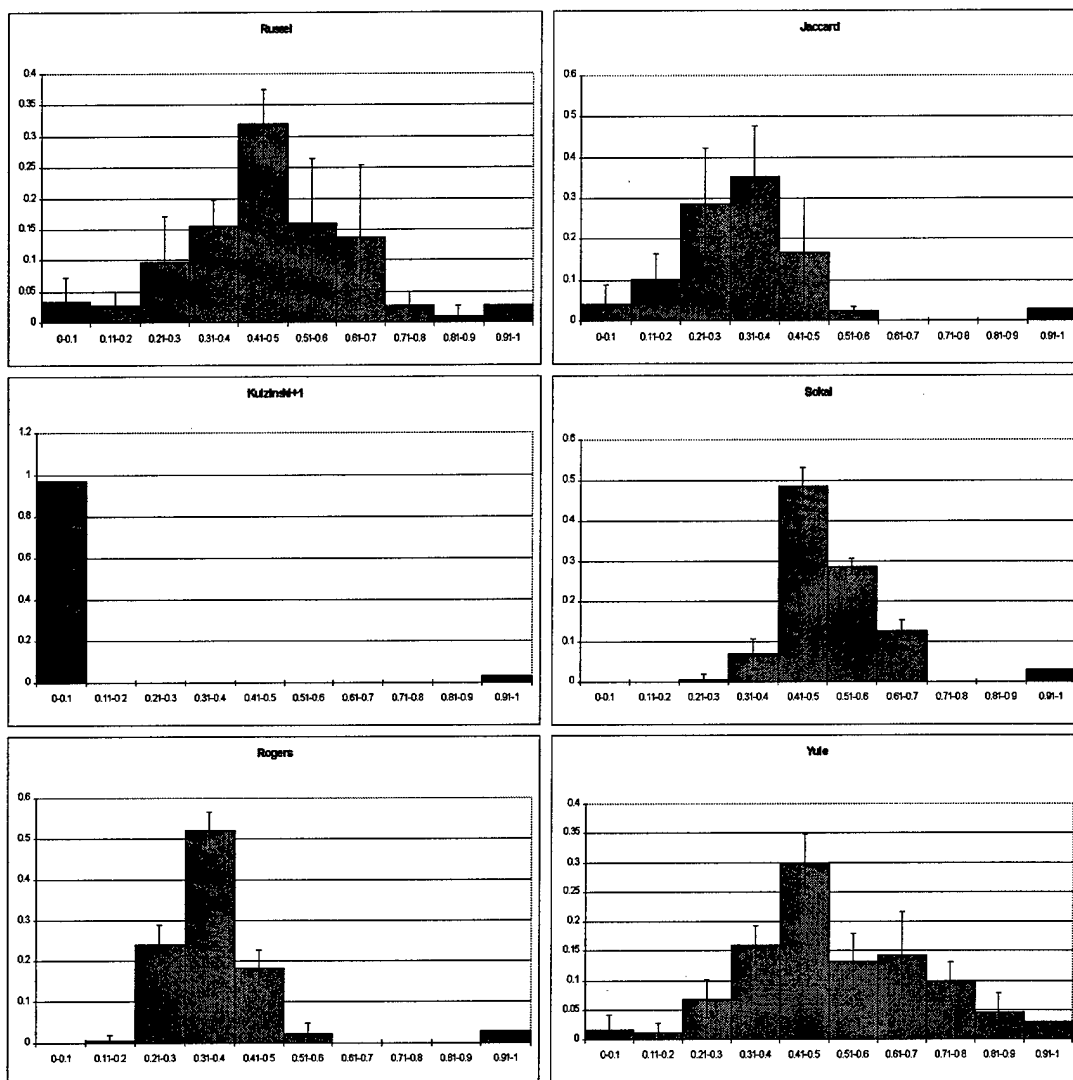
Figure 25. Normalized matching rule distribution functions part II.

ming distance in terms of specificity. These two also have large gaps in their frequency distributions between an exact match of 1 and the values of lesser matching affinity. The Rogers measure is the best of the two because of its low variance in frequency values. The correlation coefficient squared produces a close to uniform distribution, which would scale well in sensitivity, but it has a very high false positive frequency. This false alarm rate renders the correlation coefficient unacceptable. For these reasons, the Rogers and Tanimoto measure is the best choice. Its density function is a fairly good approximation

of the ideal case, but its greatest deficiency is the gap between a positive match and the next lowest frequency band. This either needs to be accounted for with a scaling of the threshold reduction, or it allows for a sensitivity gap where the threshold would have to be reduced 40% before additional sensitivity is encountered. Heightened sensitivity could also be gained by the replacement of the Rogers function with the Sokal function. This would give the system the same performance as the Hamming distance if more generality is required.

3.3.4 Conclusion. The Rogers and Tanimoto similarity measure is the best matching rule for this application. It provides a good compromise between a specific versus a general detector and can also accommodate increased sensitivity through detector threshold reduction, although a fairly large reduction is required.

For each of these matching rules, additional mathematical operations such as squaring, scaling, or taking the absolute value can have a dramatic effect on the density function histogram and the signal to noise ratio. The Yule discriminator produces a value between -1 and 1. Scaling with the absolute value produces a density function that almost exactly matches the ideal case. However, this folding of the density values about the origin produces invalid results because a value of -1, the result of two completely dissimilar strings, then becomes equal to an exact match. Other items to consider are the matching methodology and the sensitivity of the measures. The r-contiguous bits measure is highly sensitive to bit changes near the middle of the string, while less sensitive at the outer edges. One bit flip in the middle can cut the measure's value in half, while an end bit change only decreases the measure by 1. Finally, the matching methodology, whether block compare or sliding window, can produce very different results. By only comparing in successive N-bit blocks, information is lost (Figure 26). Most of the functions completely miss the exact match at position 14 because it is sandwiched between two successive 4 byte blocks. The correlation coefficient comes close due to a false positive match at position 16. The block compare methodology would only be useful in RISC processors where instructions and data are aligned on predetermined boundaries. The chunk size would have to be exactly matched to the processor word size to be effective. However, in CISC machines (those

61

running Microsoft DOS and Windows variants are host to the greatest number of viruses) instruction length is variable. Therefore, using a block compare strategy would miss important instruction and data structures. For this problem domain, the best approach is the use of the Rogers and Tanimoto matching rule with a sliding window comparison strategy.
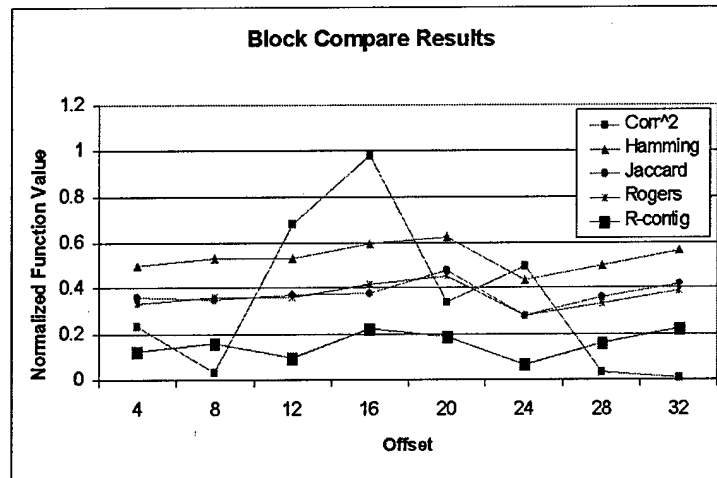
Figure 26.   Output values using a four byte block comparison.

3.4   Immune System Model Development

3.4.1   Biological Immune System Features.   The components, processes, and results of the biological immune system highlight it as an effective model for self defense. It is desirable to construct a computer virus immune system based on this model in order to overcome the reactive, non-adaptive, centralized, and monolithic nature of current anti-virus solutions. But, the fundamental differences between biological and digital systems make a mapping between these domains difficult. In order to construct an effective isomorphism, the following features, functions, and organizing principles (SHF97, MVHL99) of the biological immune system must be understood:

Parallel and Distributed: The immune system is a massively parallel architecture with a diverse set of components. These components are distributed throughout the body and communicate through chemical signals.

Multi-layered: No single mechanism offers complete immunity. Each layer operates independently, yet also in concert with all the other components, to provide a defense in depth.

Autonomous: Each entity of the immune system operates under independent control. There is no central authority and hence no single point of failure. The multitude of independent agents work together and what results is the emergent behavior of the immune system.

Imperfect Detection: A detection event does not require an single exact match, but rather, the exceeding of an affinity threshold. Imprecise detectors allow for generality in the matching process, which further allows each detector to cover a larger subset of the non-self space.

Safety: The system contains checks-and-balances, such as costimulation and activation thresholds, to ensure that detection errors are minimized.

Diversity: Diversity in the composition of each individual's immune system ensures that the entire population does not succumb to the same single pathogen. Additionally, each immune system cell only carries one form of detector. A large population of cells with a diverse set of receptor types enables the body to cover a large portion of the non-self space.

Resource Optimization: It is combinatorically expensive and too resource intensive to maintain a complete set of non-self detectors. Through the use of programmed cell death and cell division, the system maintains a random sampling of the search space at any one time.

Self/Non-self detection: Through non-self receptor death and generation, the immune system has the ability to detect and respond to the presence of pathogens, even those which have not been encountered before.

Selective Response: After a detection, chemical signals and the identification method effectively classify the antigen. This determines the exact response to an infection.

Memory: Memory B cells enable the immune system to "remember" past infections and prime the system for an improved response upon later infections by the same or similar antigen.

Adaptive: The system evolves through clonal selection and hypermutation to improve the antigen recognition capabilities and therefore improve the overall system performance.

3.4.2 Artificial Immune System Model. At a high level of abstraction, the main structures of the immune system map logically into information system entities (Figure 27). The biological immune system correlates to a CVIS, whose function is to detect and eliminate computer virus pathogens. These antigenic programs are made up of symbolic string (i.e. bits, bytes, or words) patterns (Section 3.2) that detection algorithms search for by employing pattern matching functions (Section 3.3).

Immune system   ⟶   CVIS

Pathogens (antigens)   ⟶   Computer viruses

B-cells, T-cells, and antibodies   ⟶   Detectors

Proteins   ⟶   Strings

Antibody/antigen binding   ⟶   Pattern matching
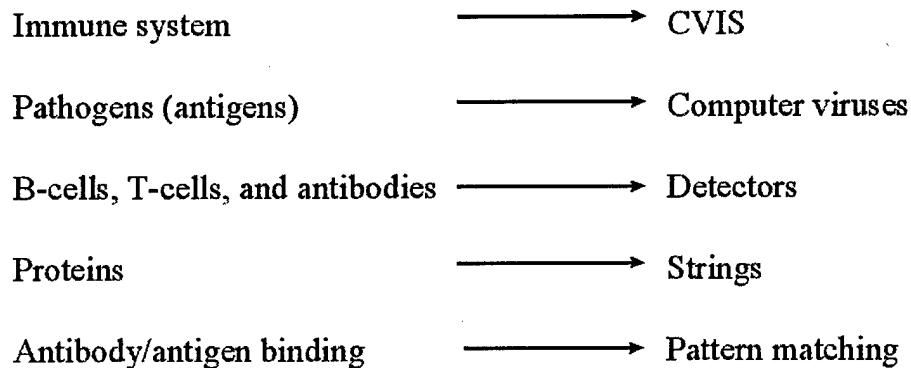
Figure 27. Biological to computational domain top level mapping.

The previously identified biological immune system features, functions, and organizing principles are further decomposed into lower-level information system entities and operations. The mapping between these functions and organizing principles can be seen in Table 4. The autonomous, multi-layered, and distributed features of the biological immune system intimate a distributed multi-agent system utilizing a diverse array of agent detectors. These detectors maintain "antibody" search strings that are censored at creation via the negative selection algorithm. The detectors are deployed with a pattern match-

ing function that produces a relative affinity based on the similarity of the antibody and antigen strings (Section 3.3.1).

If a detector exceeds an affinity threshold, then it is activated. If multiple antibody strings are activated, "affinity maturation" is used to maintain only those detector strings that best match the malicious code. This process and the "programmed cell death" of non-activated strings results in a continual searching of the non-self space along with a retention of only the best matching antibody strings. A match that exceeds the affinity threshold also requires a costimulation signal in order to reduce false positive errors. A confirmed valid detection results in a selective response that utilizes the best means available, either repair, deletion, or quarantine. A repair can occur if an exact classification of the infecting virus can be made and a known "antidote" algorithm is available. Otherwise, the infected file must be deleted, or immobilized (quarantined) in order to not pose a risk to the infected system or its neighbors.

3.4.3 System Logical Hierarchy. The deployment of an agent-based CVIS should be distributed with redundant links and no centralized control in order to realize the fault tolerance and no single point of failure present in the BIS. However, a logical system hierarchy is require to apportion functional, management, and reporting tasks. These levels facilitate the dissemination of preventative information as well as the recognition and early suppression of computer virus epidemics (Figure 28). The hierarchy is divided into the system, network, and local levels that map to a larger biological abstraction (Table 5). The assignment of functionality to the three layers borrows from the structure and operation of the self-adaptive CVIS (LMV99) and the Computer Health System (CO99).

System Level – Provides health status of the community.

– Identifies problems, durations, trends, and locations.

– Promotes system health awareness by providing prevention information and sharing community status, thresholds, and vaccinations.

– Provides a global storehouse for memory detectors.

Network Level – Focuses on the local community of machines.

– Sets system priorities by controlling activation thresholds and system

Table 4.   Biological to computation domain mapping (SHF97, MVHL99).

Immune System	Information System
Parallel & Distributed	Distributed system software utilizing data network communications. Detection and elimination activities operate in parallel.
Multi-layered	Multiple detector types monitor various input sources (email, file system, boot sector, etc.). Policy guidance in order to implement barriers to initial infection (regular vaccinations, no executables in email, etc.).
Autonomous	A multiagent system of autonomous software agents.
Imperfect Detection	Detectors utilize partial string matching functions with an activation threshold.
Safety	Costimulation through detection alarm validation to reduce false positive errors.
Diversity	Each detector node generates a statistically unique set of non-self detectors.
Resource Optimization	The detector set repertoire is continually resampled by reinitializing detector strings that are not activated within a certain time frame.
Self/Non-self Detection	Utilize the negative selection algorithm to censor detector strings so that only non-self patterns remain. Employ these patterns through input source scanning.
Memory, Adaption	Retain detector strings that effectively match non-self. Upon multiple separate detector matches, only retain those with the highest affinity.
Selective Response	Eliminate malicious code by the best means available, such as repair, deletion/replacement, or quarantine.

responses.

- Collects local system status.

- Reports local status to the system level.

- Dispenses vaccinations and preventative information.

Local Level – Responsible for detection, response, and memory.

- Implements innate and acquired immunity through self/non-self detection.

- Generates infection warnings.

- Implements anti-virus responses.

- Implements local memory.

Table 5.   System hierarchy domain comparison.

Level	Network	Biological
System	Internet	Population
Network	Subnet	Community
Local	IP	Individual
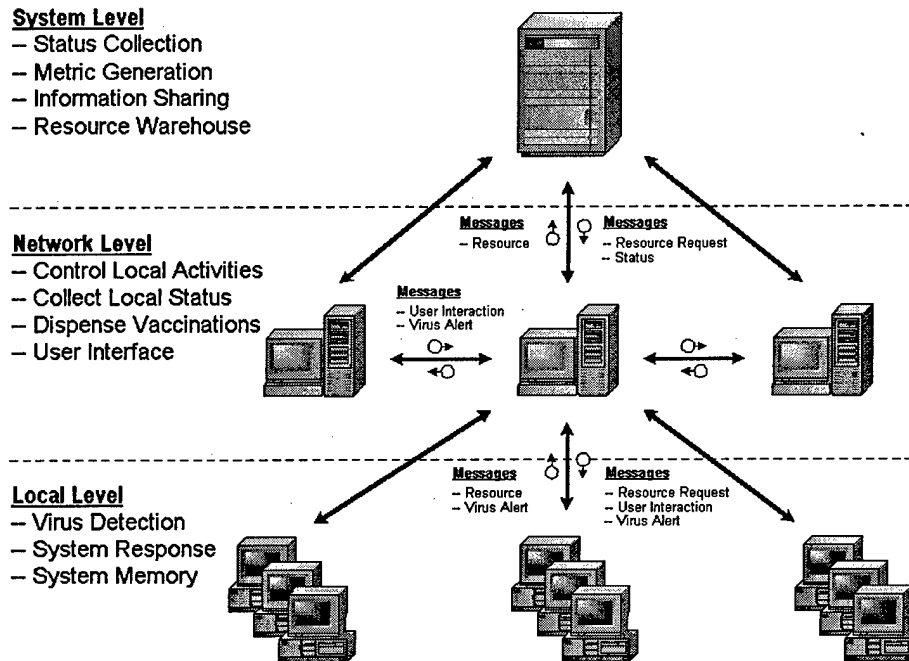
Figure 28.   Model logical hierarchy.

3.4.4   Local Model.   At the local level, detectors encompass the features of B cells, T cells, and antibodies into a unified detection entity. In order to reduce the overhead of maintaining multiple separate instances of detector objects each with a separate antigen receptor, each detector contains a set of detector strings. These strings are initially censored via negative selection and also have a finite lifetime, unless they are promoted to a memory "cell." False positive errors are reduced through an activation threshold and an external costimulation requirement. These processes infer an antibody scan string lifecycle (Figure 29).

67
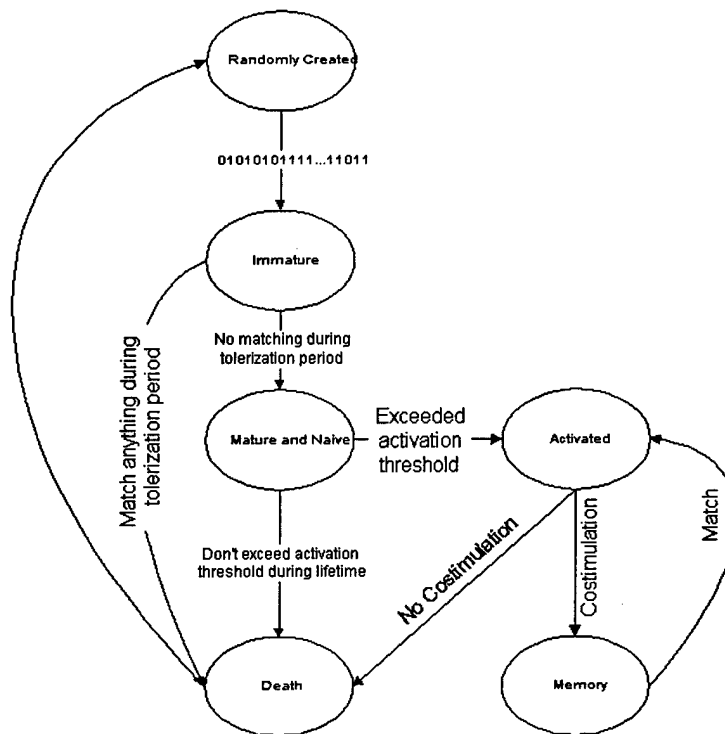
Figure 29. Detector string lifecycle (HF99).

3.5 Language Selection

The development of a CVIS is a multidimensional endeavor that cuts across several problem domains. The language chosen to implement the prototype system must be able to support these diverse requirement sources and function within the target operational environment. The overall goal is to enable the development of a distributed agent proof-of-concept prototype.

Distributed agents are the target architecture for this system (Chapter I, Sections 2.3.7, 2.4, and 4.1). To support agent development, an object-oriented language is desired. Furthermore, in order to utilize distributed, multiagent capabilities, an effective and efficient communications library, or agent development kit, must be available. This library should run on Microsoft DOS-based computers.

The most common target of malicious code attacks are Microsoft DOS and Windows based computers (Section 2.1.1). Additionally, WindowsNT and UNIX machines are often

68

Table 6. Implementation language selection summary.

Requirement	Visual Basic	C++	Java
Object-oriented	√	√	√
Network Communications	limited	limited	√
File I/O	√	√	√
MSWindows executable	√	√	√
Cross-platform capability		limited	√
Fast code		√	
Prototyping	Fast	Slow	Medium

used for file servers or network gateways for DOS clients. A language that can create executable programs for Microsoft machines is a necessity; support for other hardware and software systems is an advantage. An additional operational concern is the desire for unobtrusive operation. The CVIS system should be able to run in the background without over-utilizing system resources.

The initial system implements file infector detection routines, so the language must support file system access. Future improvements could include other detectors, such as email or network packet scanners. Therefore, the language should contain an application program interface (API) to access these input sources. Additionally, the ideal language would support fast prototyping. The overall goal is the construction of a proof-of-concept system. A reduction in the project schedule and technical risk by selecting an easy to use and fast coding language is an advantage over some of the operational requirements.

Only Visual Basic, C++, and Java are evaluated for selection. Other object-oriented or agent languages are available, such as Python or Telescript (Ret99, KT98), but in order to facilitate future understanding and maintenance of the code, a more mainstream implementation is desired. The results of the comparison, based on the previously defined criteria, can be seen in Table 6.

Visual Basic's advantage is its fast prototyping capability. It is a fourth generation language, so it is not as fast as a lower level, compiled language. It is also solely targeted to Win32 platforms, so its scope is a somewhat limiting. It does provide object-oriented features and a rich API set, including file manipulation. Conducting network operations re-

quires using the WinSock ActiveX control and specifying low-level details, such as TCP/IP address, port, and protocol. VB's fast prototyping capability does overcome the rest of its limitations.

C/C++ meets most of the requirements for the system. Network communications are accomplished through basic, and laborious, socket manipulation. Only one communications library, the Adaptive Communication Environment (ACE) (Sch99), appears to exist for C, but there are a few more ADKs (Ret99). Their documentation however, is scarce. Cross-platform portability is somewhat limited as significant portions of the code would probably have to be rewritten to conform with alternate operating system communications constructs. C's great advantage is fast code. However, this does not overshadow a slower development time combined with notoriously difficult debugging. The schedule risk and communications library uncertainty make C++ a good choice if speed was the only requirement.

Java's weakness lies in the fact that it is a slow, interpreted language. However, its advantage for this project is the extensive availability of communications and ADK solutions. In fact, Java is quickly becoming the *de facto* standard for agent-based systems. Additionally, Java strikes a balance between the flexibility of lower level code and ease of use to make it a relatively fast prototyping language. Also, because it is interpreted, Java possesses the "write once, run anywhere" property. This feature is more than is required for this prototype, but provides deployment options not available in the other solutions. Due to the extensive communications options and relatively quick prototyping, Java provides the best implementation language for this project. The poor performance, relative to compiled languages, is a technical risk in terms of usability. But, there do exist just-in-time (JIT) compilers that hope to improve Java's performance limitations. This tradeoff is evaluated as part of the system performance testing.

3.6 Summary

This chapter discusses several elements that are prerequisites for system design. The design methodology is a combination of the waterfall model and MaSE. This allows for a simplified, yet complete design process that integrates an agent-based approach. Some

of the internal agent components are explored by specifying the computer virus problem domain and then choosing a viral pattern matching function that best meets the needs of the system. Several candidates from the pattern recognition field are investigated. The Rogers and Tanimoto bit-matching function is selected based on its ability to provide an adjustable balance between specialized and general detectors. The system and local level immune models are also developed. These provide the architecture to develop a distributed agent immune system that encapsulates the strengths of the biological model. Finally, Java is selected as the implementation language due to its balance between functionality and prototyping speed. Also, there exists a wide variety of Java-based communications solutions for creating a distributed system. The following chapter discusses the actual system design based on these elements.

IV. Computer Virus Immune System Agent Design

This chapter presents the system design and implementation through an agent-oriented approach. The MaSE process is followed to produce an agent-based CVIS through the integration of the biological models. The distributed agent architecture is enabled through the design of the agent communications. Several messaging libraries and implementation details are explored for the development of the system communications backbone.

4.1 Agent-oriented Design

4.1.1 Domain Level Design. The domain level design encompasses identifying agents, their interactions, and the protocols used for this communication. The domain level design is begun through use-case modeling. Use-cases describe how the system is employed. They help decompose the system model into individual actors and objects.

- **Generate Non-self Strings**

    1. The generator creates a non-self detector string.

    2. The generator tests this string against all known self.

    3. If a match on self occurs, the string is destroyed and a new string is generated. This process is repeated until no match occurs and the string graduates to an immature state.

    4. If a detector string is set to memory type, the generator adds this string to non-volatile storage.

    5. The generator logs all actions performed.

- **Compliment**

    1. The compliment opens the input source.

    2. The compliment quickly performs pattern matching using a set of reduced length antibody strings.

    3. If a match occurs which exceeds a lenient affinity threshold, the compliment stores a pointer to the offending file.

4. After scanning, the compliment signals the detector with the set of possible infections.

5. The compliment logs all actions performed.

- **Detect Foreign Bodies**

  1. The detector opens the input source.

  2. The detector performs pattern matching using one or more generated strings.

  3. If a match occurs which exceeds the affinity threshold, the detector raises a warning and stores a pointer to the offending file.

  4. After a designated time period, if a detector string has not been elevated to a memory type, the detector destroys the detector string and signals the generator to generate a new one.

  5. The detector logs all actions performed.

- **Monitor Warnings**

  1. The monitor coordinates the activities of the local agents.

  2. If a warning message is received, the monitor raises an alarm and signals the helper.

  3. If an alarm is received from an adjacent monitor, the local monitor decreases the local activation threshold.

  4. The monitor communicates the local status to the controller.

  5. The monitor logs all actions performed.

- **Costimulation**

  1. If an alarm is raised, the helper reports the alarm and asks for costimulation.

  2. If no costimulation is received, or a negative costimulation is received, the helper signals the detector to destroy the detector string.

  3. If costimulation is received, the helper signals the classifier and signals the detector to graduate the detector string from immature to memory state.

4. The helper logs all actions performed.

- **Classify**

  1. The classifier gets the pointer to the infected file from the detector.

  2. The classifier compares the file data bits with known virus signatures.

  3. If a match is found, the classifier signals the repairer.

  4. If no match is found, the classifier signals the killer.

  5. The classifier logs all actions performed.

- **Remove Foreign Body/Kill**

  1. The killer notifies the helper that no known cure is available.

  2. The killer asks the helper to confirm the deletion of the infected file.

  3. If a confirmation is received, the killer deletes the file.

  4. If no confirmation is recieved, the killer asks the helper to confirm the quarantining of the malicious code.

  5. If no confirmation for quarantine is received, the killer warns the administrator of the presence of active malicious code on the system.

  6. If confirmation for quarantine is received, the killer moves the infected file to a safe location and renders it unexecutable.

  7. The killer logs all actions performed.

- **Repair**

  1. The repairer notifies the administrator that a known cure is available.

  2. The repairer asks the administrator to confirm the application of the repair.

  3. If a confirmation is received, the repairer repairs the file.

  4. If no confirmation is recieved, the repairer asks the helper to confirm the quarantining of the malicious code.

  5. If no confirmation for quarantine is received, the repairer warns the administrator of the presence of active malicious code on the system.

6. If confirmation for quarantine is received, the repairer moves the infected file to a safe location and renders it unexecutable.

7. The repairer logs all actions performed.

- **System Control and Reporting**

1. The controller provides metrics to the administrator on system operation.

2. The controller provides the health status of the community.

3. The controller provides preventative information to the monitors.

4. The controller coordinates information passing between non-local monitors.

5. The controller logs all actions performed.

In the strict use-case methodology, actors are anything that communicate with the system, but that are external to the system itself (Pre97). However, because this is a system of autonomous agents, each agent also becomes an actor and the notion of a system is somewhat nebulous. The system is the sum total of all the parts and their actions. The result is a complete system architecture based on the biological model and the logical system hierarchy (Section 3.4.3, Figure 28). The interactions between actors/agents is better visualized with a use-case diagram (Figure 30). The use-case diagram is further used to define conversations between the identified agents.

Figure 30. Use-case diagram.

The design of agent types is completed by decomposing the use-cases into individual agents. A base set of seven agent types are identified and the mapping of use-cases to agents can be seen in Figure 31.

Generate Non-self Strings ⟶ Antibody

Detect Foreign Bodies ⟶ Detector

Monitor Warnings ⟶ Monitor

Costimulation ⟶ Helper

Classify ⟶ Classifier

Remove Foreign Body/Kill

Repair ⟶ Cleaner

System Control & Reporting ⟶ Controller

Compliment ⟶ Compliment
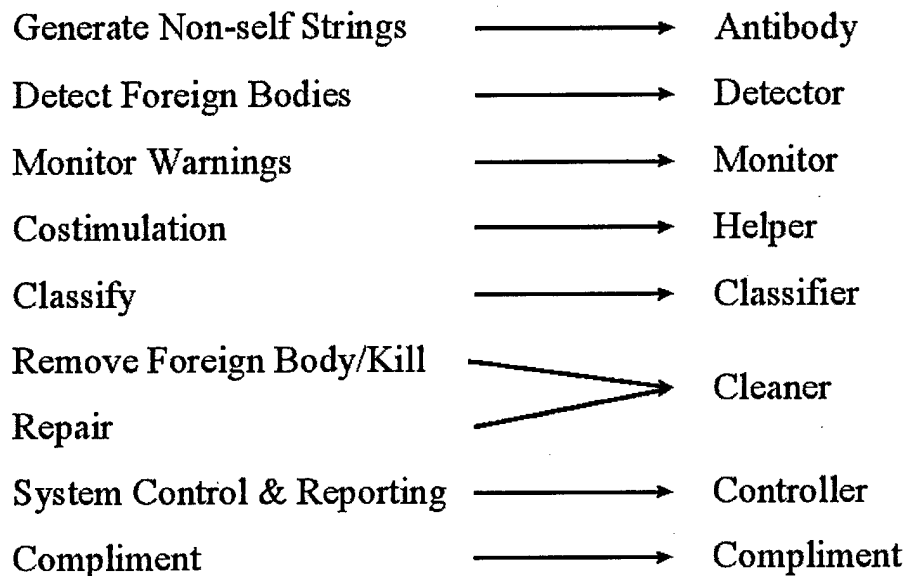
Figure 31.   Decomposition of use-cases to agents.

The Antibody agent encapsulates the generation and maintenance of search strings. The Detector agent uses the services of multiple Antibodies in order to scan an input string for malicious code. The Monitor controls the local area detection thresholds, communicates with the controller and other local monitors, and generates alarms to be acted upon by Helper agents. Helpers perform the tasks of interfacing with the administrator, such as soliciting costimulation in order to overcome the problems of imperfect detector strings. Classifiers identify the exact infector responsible and send the appropriate cleaner to fix the problem. Cleaners remove the virus from the system using the best means available, repair, deletion, or quarantine. The definition of agents types concludes with assigning goals and services to the individual agents.

The agents with their goals and services can be seen in Table 7. The services provided by the agents are requested through interactions with other agents. These interactions are carried out by message passing "conversations."

Table 7.   Agents, goals, and services.

Agent	Goals	Services
Antibody	Generate, maintain, and store valid scan strings	Generate Graduate to memory Destroy scan string
Detector	Detect malicious code at the input source	Scan input source Receive vaccination Update detection threshold Destroy antibody string Graduate antibody string to memory Get pointer to input source
Compliment	Quickly scan input for malicious code	Scan input source
Monitor	Coordinate the actions of a local neighborhood of agents	Receive information from a Controller Send, process, and receive alarm messages Receive warning messages Update detector detection thresholds
Helper	Communicate with the system user/administrator	Receive system information Receive costimualtion Receive action confirmation
Classifier	Implement the system response to an infection	Identify infection agent
Killer	Remove viral infections	Delete infected input
Repairer	Repair viral infections	Repair infected input
Controller	Coordinate global system operation Generate system operation metrics	Receive monitor status messages

4.1.1.1   Agent Conversations.   Agent "conversations" define possible interactions between agents (DeL99a). Conversations are used by an agent to request the services of another in order to fulfill its goals. Through the coordinated use of each other's services, the CVIS as a whole is able to detect, identify, and remove malicious code from the system. The required coordination is accomplished through conversations.

The conversations are developed from the use-cases. Interagent interactions are described in the use-cases and shown as links on the use-case diagram (Figure 30). Each link becomes a conversation, or part of a more complex interaction. The use-case interactions generate the conversations shown in Table 8.

The agents (Figure 31) and conversations (Table 8) derived from the use-cases are combined using object-oriented modeling into an agent diagram (Figure 33). Within the diagram, agents are modeled as objects and their conversations become associations. These conversations are designated using a nomenclature prefaced by a lowercase 'c.' The associations are also represented by a conversation object within the system conversations package.

78

Table 8. Agents and their conversations.

Conversation	Initiator	Receiver	Description
cRaiseWarning	Detector	Monitor	Notify of a possible viral infection.
cUpdateThreshold	Monitor	Detector	An infection has occurred in an adjacent node, reduce the detection threshold to increase awareness.
cCostimulation	Monitor	Detector	A warning has been validated (not validated).
cVaccination	Controller Monitor	Monitor Detector	Vaccinate detectors with this string.
cSendMessage	Controller, Classifier, Monitor	Helper	Notify the System level administrator with the attached text.
	Controller	Monitor	Notify the Network level administrator with the attached text.
cStatus	Controller	Helper	Display system status to the administrator.
	Monitor	Controller	Send the Network level status to the System level.
cRaiseAlarm	Monitor	Helper	Ask for costimulation.
cConfirmation	Classifier	Helper	Ask for verification of virus removal actions.
cPassFilePointer	Detector	Classifier	Pass the Classifier the infection location.

By further modeling each conversation as an object, the power of object-orientation can be used to the systems's advantage. All the explicit conversations types inherit from a base conversation class (Figure 32). This allows common communication routines to be reused in all conversations. Additionally, these conversation objects become the components, internal to the agents, that create communication connections with other agents. Within each conversation, a communication protocol is defined. This protocol is made up of the messages required to exchange information, requests, and transactional handshakes between agent services.
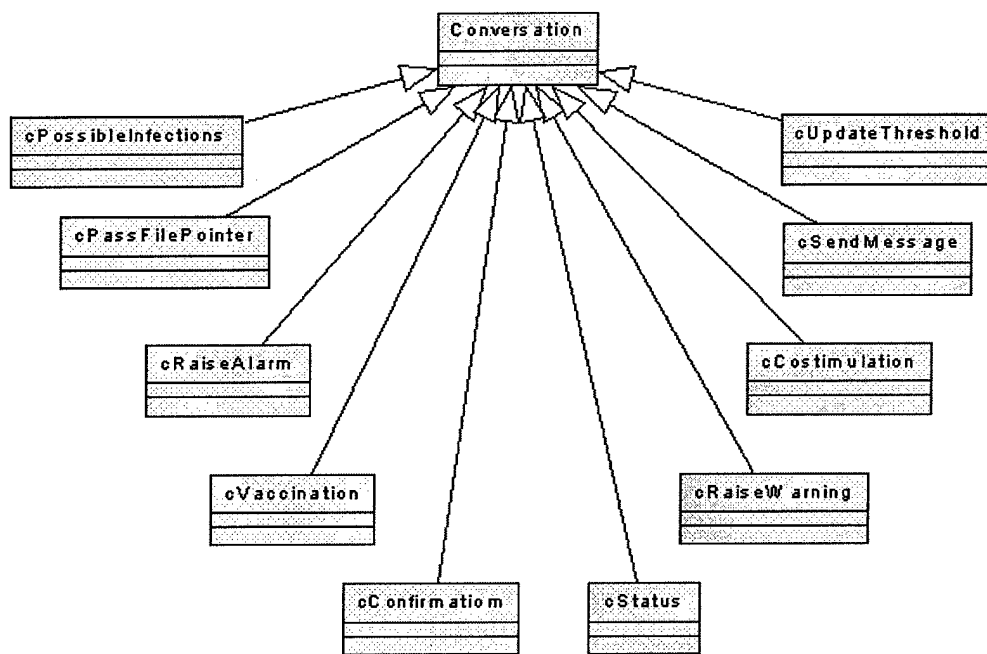
Figure 32.   Agent conversation hierarchy.
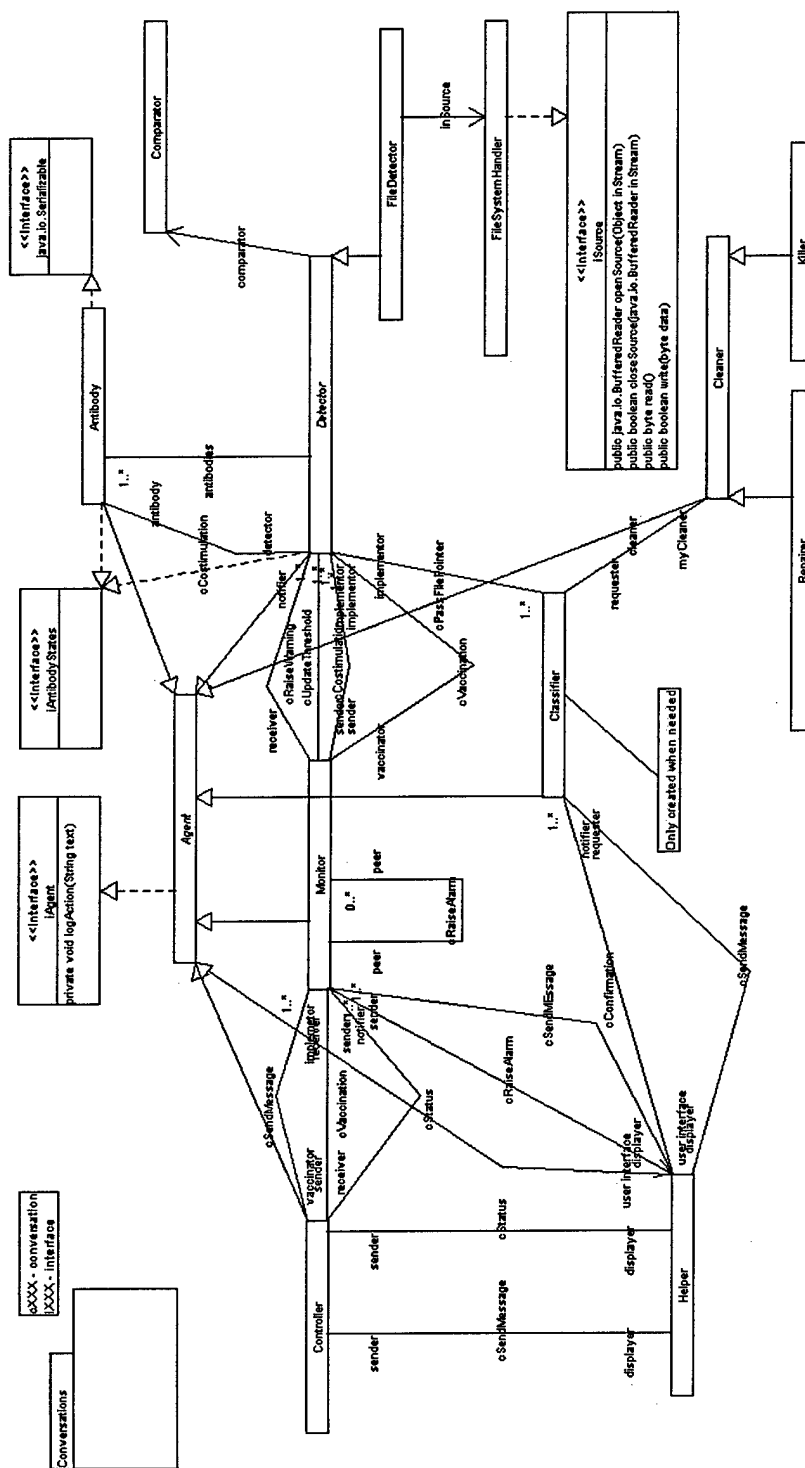
Figure 33.   Agent class diagram (See Appendix B for large format availability).

4.1.1.2 Conversation Protocols. The final step in the domain level design is the definition of the protocols used in agent conversations. Conversations define an inherent message passing convention, agreed to by all agents that are party to the communication (DeL99a). The MaSE methodology defines the use of state transition diagrams (STDs) to describe the conversation protocol. This actually requires two separate diagrams per conversation; one to describe the agent states involving the initiator of the conversation and the other for the receiver. The STDs connect message send and receive events to the internal agent operations. The agent services become agent object methods called within the conversation states (Figure 34). The remaining STDs can be found in Appendix A.1.1.
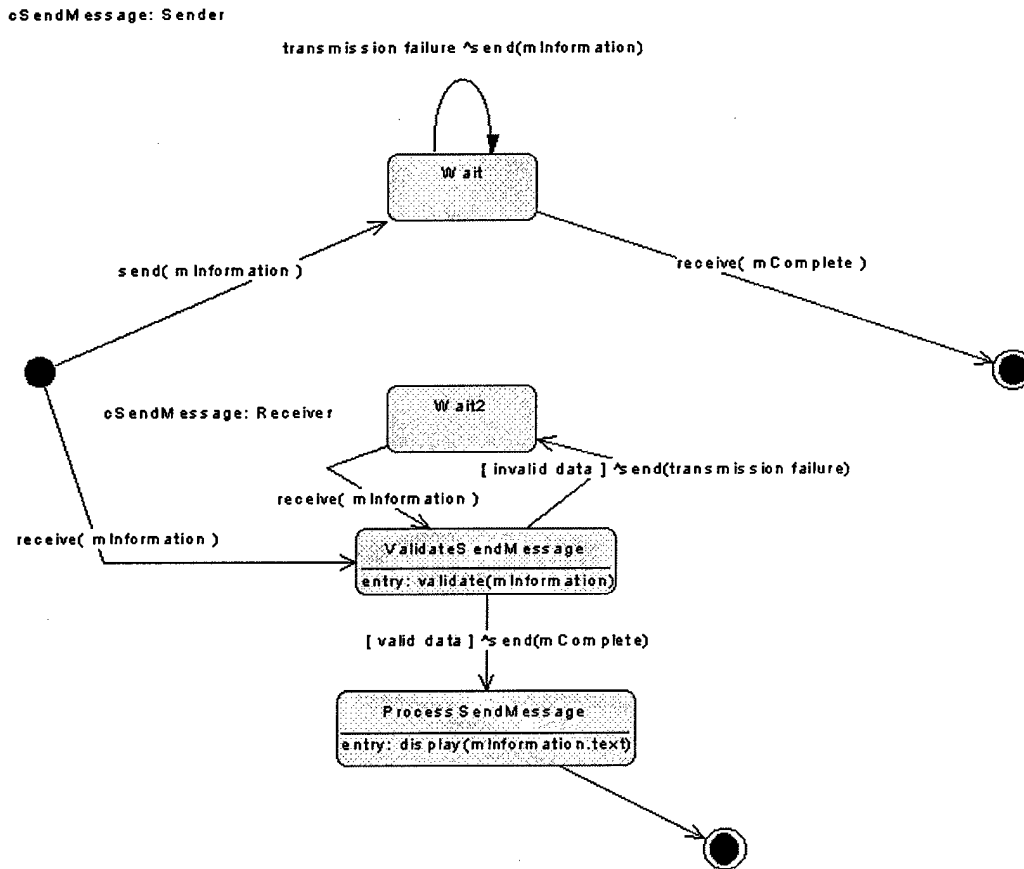
Figure 34. Agent message passing conversation state transition diagram.

These diagrams provide the connection between message receipt and agent actions, however missing from the STDs is a clear picture of a conversation's temporal relation-

ships. This is especially important for designing protocol message sequence handshaking. Therefore, message sequence charts are used in addition to STDs to more completely describe the agent conversation design (Figure 35). The remaining charts can be found in Appendix A.1.2.
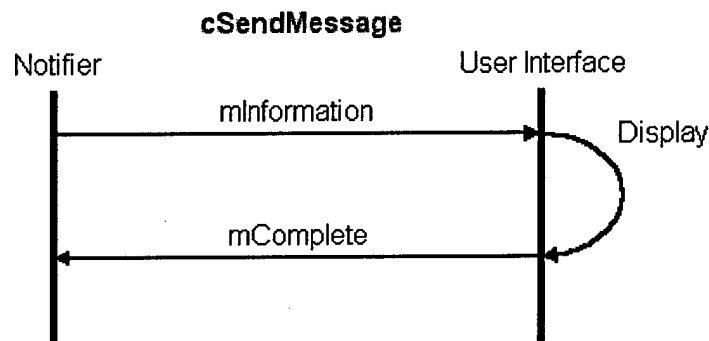
Figure 35.   Agent message passing conversation message sequence chart.

The complete design of the agent conversations through the use of state transition diagrams and message sequence charts completes the domain level design. The next design step is the agent level design. This stage involves mapping the actions called out in the STDs to internal agent components (DeL99a).

4.1.2   Agent Level Design.   Agent level design encompasses designing the lower level details within each agent. This includes the definition of internal agent components and data structures required to support the defined conversations (Table 8) and services (Table 7). The components, methods, and attributes are suggested by the actions within the STD states. Additionally, other internal processing is required to support the complete set of agent services. The derived components, operations, and attributes can be seen in Table 9. These are added to the agent object diagram (Figure 36). The low level details of the components and methods are refined in the MaSE component design step.

Table 9.   Agent components, methods, and attributes.

Agent	Components	Methods	Attributes
Agent (*abstract*)		logAction(String text)	.
Antibody		setState(int) generate()	final int immature = 0 final int naive = 1 final int activated = 2 final int memory = 3
Detector (*abstract*)	Conversation Vector antibodies	vaccination(String pattern)	final int maxAntibodies float activationThreshold final float thresholdIncrement final boolean up = true final boolean down = false
FileDetector	FileSystemHandler	negativeSelection(Antibody) Vector scan() updateThreshold(boolean direction) destroyDetectorString(String pattern) graduateToMemory(String pattern) save(Antibody)	
Monitor	Conversation	display(String text)	
Helper	Conversation	process(String text) display(String text) boolean getUserInput(String question)	
Classifier	Conversation FileSystemHandler	classify(Tag target)	
Controller	Conversation	createMetrics() createStatus() process(String text) updateStatus() publish(String text)	

Figure 36. Complete agent class diagram.

4.1.3 Component Design. This prototype represents a first-of-a-kind system. Because of this, there are not any significant preexisting components available for reuse. This requires developing all the algorithms from scratch. Most of the operations require only common data processing, such as opening and closing files. However, the matching function, negative selection, and the scan operations are unique to this system. Additionally, the data structures required within each of the conversation messages needs further definition. The system source code contains all other details (Appendix B).

4.1.3.1 Comparator. The comparator object contains a compare method that takes in two strings and returns the value of the Rogers and Tanimoto matching function. The abstract Detector agent contains a comparator component that derived classes, such as the FileDetector agent, utilize to perform their scanning and negative selection routines.

Java does not contain an API to directly manipulate the individual bits within a byte. Such operations are required in order to calculate the a, b, c, and d values for the Rogers similarity measure (Section 3.3.1.2). A function is available that turns bytes into bit strings. This string must then be indexed "bit" by "bit" in order to perform the Rogers calculations. All of the required string operations in this methodology result in very poor performance. So, an improved algorithm is implemented.

The bitwise compare method utilizes a rotating byte mask with all zeros, except for a single "1" bit. This mask is used with the bitwise AND function to pick off the individual bit values in the input strings (DD98). This bitwise operation runs significantly faster than the previous string conversion and comparison method. The improved algorithm is placed in a BitwiseCompare object that is inherited by the comparator component. This component is used by the FileDetector agent for scan and negative selection operations.

4.1.3.2 Scanning. The scan and negative selection methods operate almost identically. Both are methods of the FileDetector agent, as opposed to the abstract Detector. This is due to the dependence of these operations on the designated input stream.

Any future detectors for other input streams would need to implement their own unique versions.

The scan method gets each byte of the file system, adds it to the sliding window, and compares the window against all antibody strings. If the matching function return value exceeds the detection threshold, the antibody and the offending file are added to a linked list for further processing (costimulation, identification, and repair). This prototype has a hard coded scan directory. This simplifies testing, but future improvements could easily make this a user selectable option.

The negative selection algorithm operates essentially the same as the scan, but it contains a recursive element. Negative selection is sequentially called on each antibody after instantiation. On a self match, the antibody string is regenerated and negative selection is called again. The recursion bottoms out after an antibody successfully completes censoring. Negative selection utilizes the same hard coded scanning directory and known self is hard coded as files in that directory beginning with "SELF". The final components elaborated in this section are the conversation messages.

4.1.3.3 Messages. The message sequence charts and the conversation state transition diagrams define individual messages that are passes between agents. The message payloads are defined by the conversations and the data that must be transformed within the agent states (Table 10). The contents of several message types are similar, so object-oriented inheritance along with abstract message types are utilized in order to maximize code reuse. The complete messages and their relationships are shown in the message hierarchy diagram (Figure 37).

Table 10.   Conversation messages.

Message	Contents
mTargetList	Vector of filename
mRaiseWarning	filename, String pattern
mRaiseAlarm	filename
mValidAlarm	filename
mInvalidAlarm	filename
mPassFilePointer	filename
mCleanTarget	filename, VirusID, ActionID
mConfirmation	String text
mActionConfirmed	boolean
mComplete	
mDestroyDetectorString	String pattern
mGraduateToMemory	String pattern
mUpdateThreshold	boolean direction
mStatus	String text
mInformation	Sting text
mVaccination	String pattern
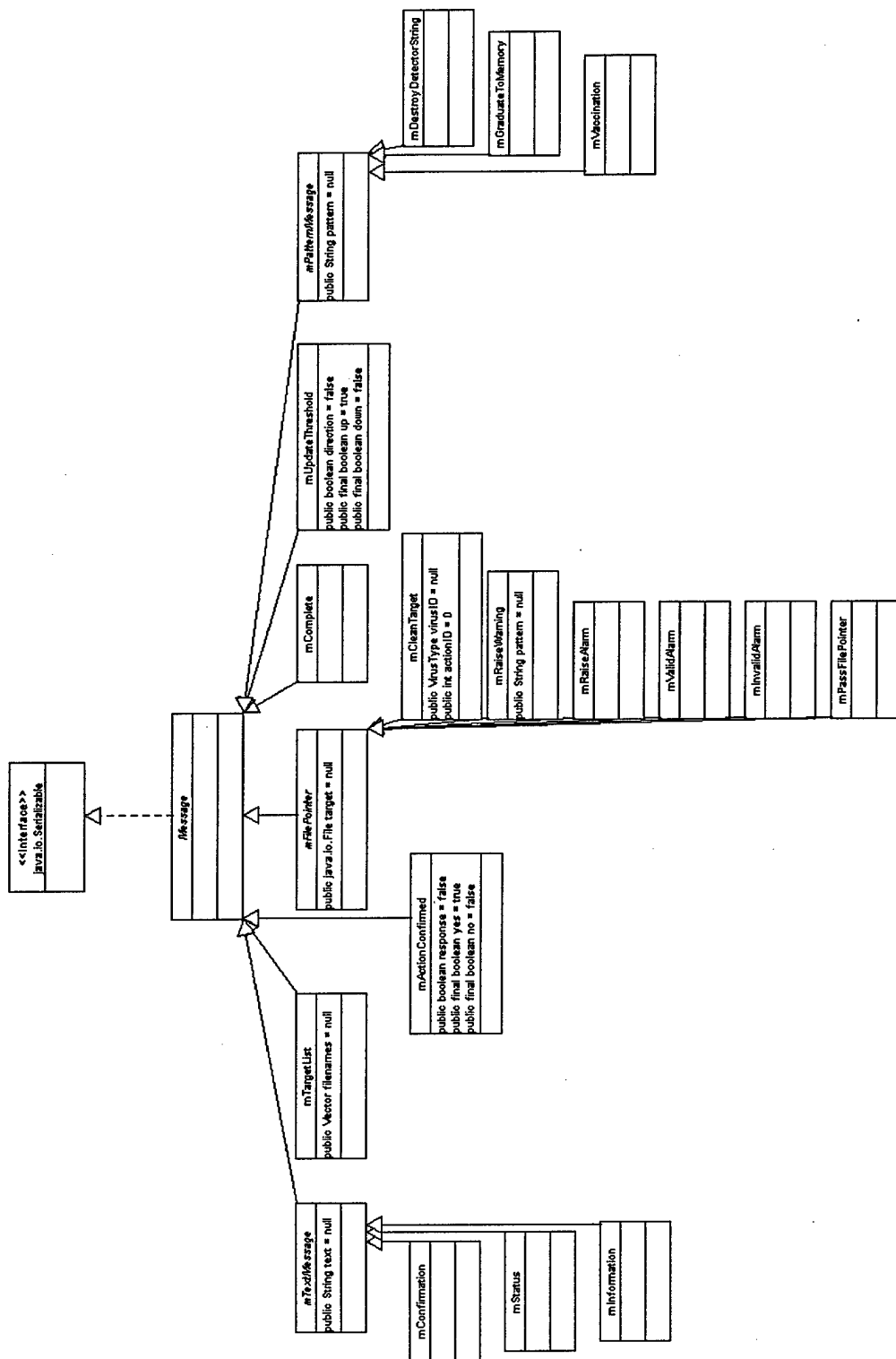
Figure 37. Message hierarchy.

89

The exact processing within the agents is initiated based on the type of message received. In Java, each object inherently knows its type. So, type checking is used within the conversation logic to parse messages. This methodology also leads to message types that contain no data. Their unique type is used for processing. With the low level agent components specified, the final design step is to construct a complete multi-agent architecture based on the individual agent types and the conversations. This task is completed in the system design.

4.1.4 System Design. The system can be defined as a set of any number of different agent types (DeL99a). The minimal set would be a Monitor and a Detector. However, a realistic system would include multiple instances of all the agent types running on distributed nodes.

The efficient mapping of agents to physical machines requires the considerations of parallel algorithm design. This is accomplished through two major components, the identification of parallel components and the mapping of tasks to processors to minimize communication (KGGK94). The division of tasks into those that can operate concurrently occurs as part of the agent decomposition (Section 4.1.1). The second part of a good parallel agent deployment is the consideration of communications costs. Detectors need to be local to their file system to avoid passing large amounts of data (conceivably the whole disk) across a network. Detections are rare, so Detectors run locally and send messages to their associated Monitors in order to minimize network traffic. Due to I/O considerations, the Classifiers and Killers should also be located with the infected file. These agents perform file operations, which induce considerable network loading if done remotely.

A major system consideration is the need for low resource overhead. The CVIS should be unobtrusive to the user. Because infections and detections are rare, Helpers, Killers, and Cleaners are not used often. Therefore, in order to not waste CPU cycles or memory on busy waiting, these agents are instantiated only when the need arises. Helpers only send messages and perform costimulation. It is logical that they be co-located with the Monitor for simplified user interaction. All these considerations are embodied in the physical deployment diagram (Fig 38).

90
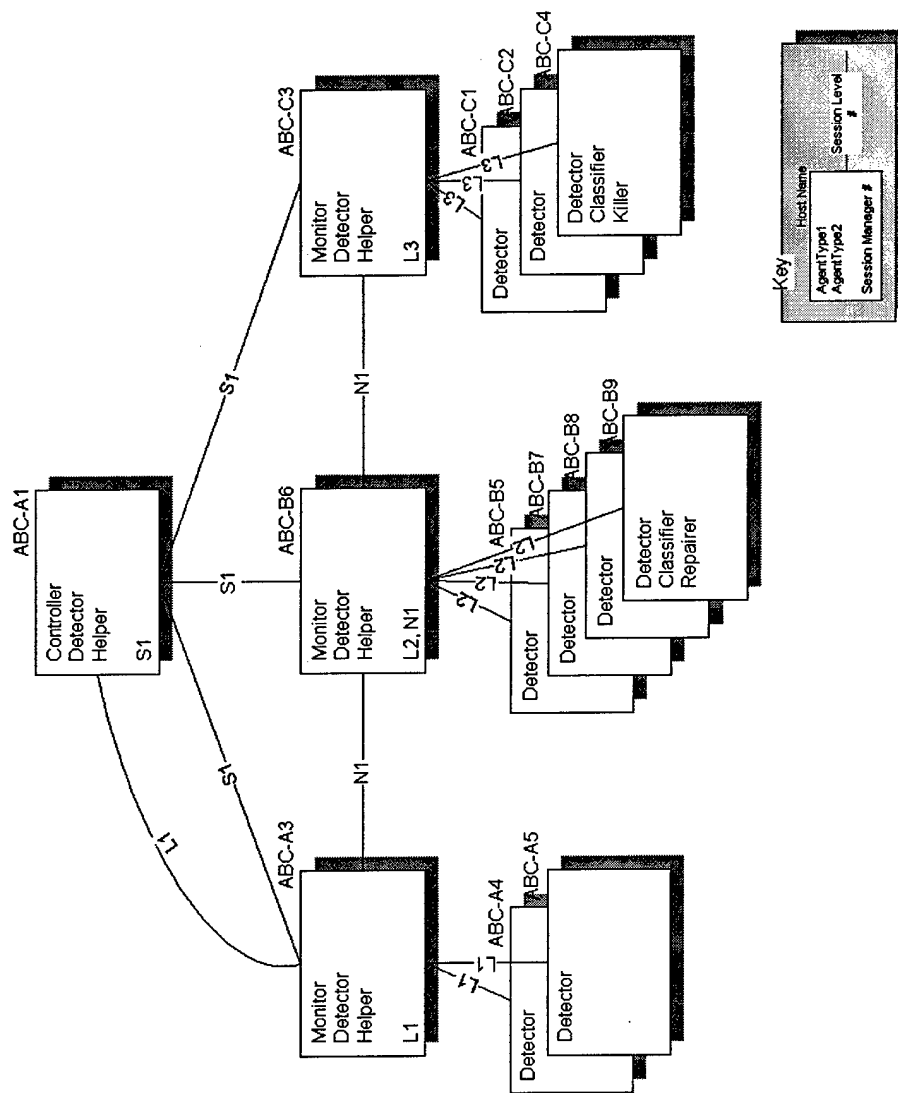
Figure 38.   Agent deployment diagram.

4.2 Agent Communications Selection

This CVIS is designed as a multiagent system. These autonomous system entities collaborate with each other in order to produce an immune system behavior. This collaboration that is inherent in distributed multiagent systems requires the use of a network backbone and a communications software layer. The Java language was chosen for this project because it was designed to operate over networks, and hence provides comprehensive network communication support (Section 3.5). However, low level TCP/IP socket construction, manipulation, and optimization is not the goal of this research. In order to develop the distributed agent-based CVIS prototype, a communications library that abstracts away the low level details of network comm is desired. There are many approaches to this problem of distributed computing including shared memory, message passing, distributed objects, and even agent development kits (ADKs). This section investigates several candidate packages for the immune system agent communications layer.

4.2.1 System Requirements. The ideal communications library would provide an efficient abstraction above the low level implementation issues while supporting the needs of agent collaboration, the immune system model, and the desire for fast prototyping. First and foremost is the need for low startup and transmission overhead. The bottleneck in many distributed applications is the communication time. Proper parallel algorithm design is accomplished through the identification of parallel components and the mapping of these entities to processors in order to minimize communications (KGGK94). This is accomplished as part of the system design and deployment (Section 4.1.4). In order to further minimize comm costs, or to not undo the efforts of effective agent decomposition, an efficient communications library is required. The needs of the agent design require the use of one-to-one and one-to-many send routines. For example, vaccinations should be broadcast to all the Detectors, while virus detection warnings need only be sent from a single Detector to a Monitor. A messaging system that only provides one-to-one capabilities could be used by making multiple sends to a list of recipients, but this would be less efficient, especially in a LAN environment where packets are broadcast to all nodes anyway. On the receive side, asynchronous messaging is desired. An infection and later detection

of the malicious code occurs with a relatively low frequency. Responses to an infection are driven from the detection event. Therefore an asynchronous event driven messaging system is desired. Next, agents can pass these messages between each other, within possibly multiple separate conversations. A communication layer that supports multiple channels over a single connection would be ideal.

The system is designed as a collaborating federation of agents. These agents could conceivably join, or leave the group at any time, for instance if workstations were turned off at the end of the day. For this reason, it is desired that the comm library support the ability to join and separate from the system, or subscribe and unsubscribe to message passing channels.

Finally, with an eye to the future, the system should be able to incorporate a security layer. In order to make a fielded system resistant to infiltration or spoofing, encryption of messages and the authentication of agents would be required. Such features are beyond the scope of this prototype, but would be necessary in an actual deployment.

Along with this diverse set of functional requirements, a communications layer that is easy to use and understand is desired. This facilitates later understanding and expansion of the design. The complete set of all of these communications library requirements are summarized in Table 11. The following candidates, segregated by type, are investigated as possible solutions to the communications needs of this system.

Table 11.   Ideal communications library capabilities.

Requirements	Description
Efficient - low overhead	Efficient comm with low startup costs
1-to-1	Point-to-point messages
1-to-many	Multicast messages
Asynchronous	
Event Driven	Agents are notified of a message arrival
Multichannel	Agents can carry on multiple conversations on one connection
Subscription	Agents can join a channel and receive all messages
High Level Abstraction	Programmer does not have to utilize low-level networking commands
Flat Learning Curve	Simple to understand and use
Security Layer	The ability to add in security features

Shared Memory

      JavaSpaces

Message Passing

      Java Shared Data Toolkit

      Message Passing Interface

Message Oriented Middleware

      Java Message Service / Java Message Queue

      agentMOM

Distributed Objects

      CORBA

      COM/DCOM

      RMI

ADK

      Voyager

      Aglets

4.2.2 Shared Memory.

4.2.2.1 JavaSpaces. JavaSpaces is a high level library for creating collaborative applications based on a shared memory model (FHA99). JavaSpaces is a service that is part of Java's Jini technology. These spaces provide a persistent, shared object exchange location through which Java applications can cooperate. JavaSpaces are based on the tuple space concept of the Linda distributed application tool. The Linda project showed that a single shared memory area with a small set of simple operations can be used to implement many parallel and distributed applications (FH99). Additional claims are that this programming model can ease design, reduce debug time, and increase maintainability (FHA99). This is due to a very simple, intuitive API based on simple reads and writes to shared memory. In fact, interaction with a space involves only reading (copy), writing, and taking (copy with remove) (Figure 39). Processes therefore collaborate through this flow of objects in and out of spaces.
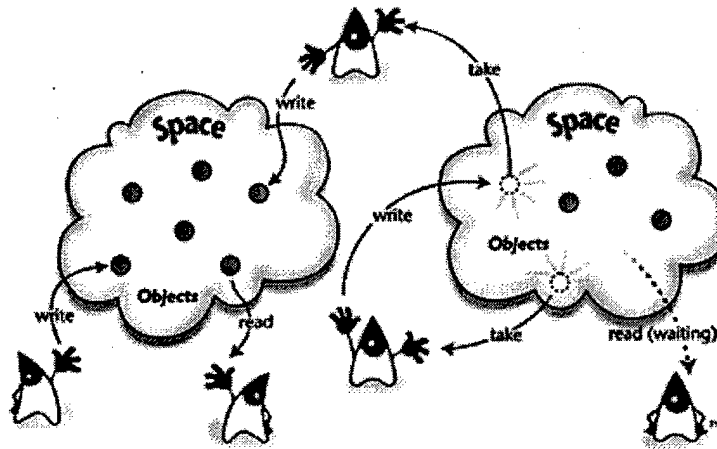
Figure 39.   JavaSpaces operation (FHA99).

The read operation retrieves objects from the space via an associative lookup. The objects in the space are compared against a provided template. An entity that matches is returned by the read operation. This implies the need for a comparison operation, probably $O(n^2)$, on every read or take, an obvious performance penalty.

The API also provides advanced features for transaction based operations on the space to ensure robustness in the face of failure. However no authentication or other security measures are incorporated. To do so would require a fairly significant rewrite of the base library classes.

The inherent nature of shared memory makes it multicast, multichannel, and asynchronous. Additionally, the space concept provides automatic load balancing among multiple consumers and eliminates the need to have a registry of participant addresses. Clients do not have to worry about multithreading, low level synchronization, or communication protocols; only read, write, and take. One-to-one messaging can be implemented through the use of appropriate pattern matching in a client take operation. Ensuring this becomes the problem of the system designer. An additional feature allows for timed leases on space entries. When the lease expires, the entry, and any copies, are deleted.

Upon subscribing to a space, a client can be notified if an entity is added, thereby providing an event model of operation. Unfortunately, all space subscribers receive this

notification, resulting in multiple unnecessary searches of the space for entries matching the respective templates.

JavaSpaces offer a very simple programming model with relatively no learning curve. They also provide a very powerful distributed programming architecture, but there appears to be a high communication cost associated with this ease and power. JavaSpaces are claimed to scale naturally, just add more clients. But, the lookup and data storage of all the entries would probably limit the scaleability. Finally, the API does not inherently support security features and adding them would be a somewhat considerable task. Therefore, JavaSpaces meet 80% of the desired requirements (Table 12).

Table 12. JavaSpaces communications library capabilities.

Requirements	JavaSpaces
Efficient - low overhead	
1-to-1	√
1-to-many	√
Asynchronous	√
Event Driven	√
Multichannel	√
Subscription	√
High Level Abstraction	√
Flat Learning Curve	√
Security Layer	

4.2.3 Message Passing.

4.2.3.1 Java Shared Data Toolkit. The Java Shared Data Toolkit (JSDT) is a communications library that is designed to support collaborative applications (Bur99). This set of classes provides an abstraction above the basic networking functionality to offer communication sessions between objects, with each session capable of supporting multiple separate data channels. The low level networking communication can utilize sockets, hypertext transfer protocol (HTTP), light-weight reliable multicast package (LRMP), or remote method invocation (RMI) for its basic connection. The exact method can be specified by the programmer during session creation. Since most of these protocols are built on sockets, it makes sense to utilize the basic socket for efficiency.

96

This architecture can efficiently support multicast messages with point-to-point being a special case. There is also support for both synchronous and asynchronous message delivery, with the latter being the default. In the asynchronous mode, a channel consumer's *dataReceived* method is called when a message arrives, thereby providing an event driven operational model. Channel consumers indicate their interest in a particular session:channel combination by subscribing to it. Additionally, the library supports managed sessions. A session manager can invite clients to join a session channel or even expel them from an existing connection. Inherent to a managed session is a security layer consisting of a challenge/reply authentication between the manager and the joining client. Additional security can be added by utilizing a secure socket layer (SSL) instead of regular, unsecure socket connections. Utilizing this capability is as simple as adding two source code lines at the beginning of a JSDT application.

The JSDT contains an extensive API with a rich feature set that appears to cover all of the desired characteristics (Table 13). Operation and implementation appear to be straight forward with a minimal learning curve. Additionally, the system architecture would scale well as new sessions and consumers can simply be added. The largest question is that of performance of the system in terms of communication time and its ability to support large amounts of agents.

Table 13.   JSDT communications library capabilities.

Requirements	JSDT
Efficient - low overhead	√
1-to-1	√
1-to-many	√
Asynchronous	√
Event Driven	√
Multichannel	√
Subscription	√
High Level Abstraction	√
Flat Learning Curve	√
Security Layer	√

4.2.3.2 Message Passing Interface.   The Message Passing Interface (MPI) is a standardized and portable communications library designed for parallel computing

applications (SOHL+96). The MPI standard defines a set of interfaces to a wide variety of message-passing communications routines. The underlying code behind the interface allows for efficient communications. Many different implementations of MPI exist for a wide variety of platforms, which facilitates code portability and operation in heterogeneous environments.

The basic API supports point-to-point and multicast messaging among groups. Messaging can be synchronous or asynchronous. All of these operations are initiated by calls that are only one layer above explicit sockets. These calls require learning a diverse set of 125 MPI functions, although a basic application requires only six of them. The standard MPI library provides bindings to these functions for Fortran77 and C/C++, but a set of Java bindings is also available (CFKL99). This communications library, mpiJava, is a Java Native Interface (JNI) connection to the C/C++ MPI library (Figure 40) (Bak98). Java applications can make full use of the complete MPI function set through mpiJava calls.
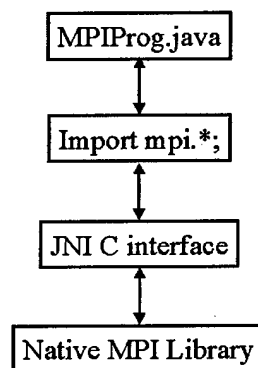
Figure 40.   MPIJava implementation layers (Bak98).

MPI can provide most of the desired communications library features, however it does not provide an event driven architecture, a high level of abstraction, or a security layer (Table 14). Any form of authentication would be difficult to implement, but message encryption could be added before sending. These problems are not insurmountable, but the biggest obstacle to the use of MPI is its focus. Agent-based systems typically operate as a client/server architecture. Agent servers provide services to agent clients. Agents differ from pure client/server environments because they typically act as both a client and a server, depending on the situation. The typical Java communications packages, such

Table 14. MPIJava communications library capabilities.

Requirements	MPI
Efficient - low overhead	√
1-to-1	√
1-to-many	√
Asynchronous	√
Event Driven	
Multichannel	√
Subscription	with Groups
High Level Abstraction	
Flat Learning Curve	
Security Layer	

as sockets or RMI, work very well for this environment. By contrast, parallel computing applications typically utilize symmetric communication (Bak98). MPI captures this idea through the use of communicators and process ranks. Processes are assigned a rank based on when they contact the communicator. This rank is then used to determine their function (Figure 41). Each process runs the same code, with internal differentiation based on the process rank. This model of operation makes MPI's use in an agent-based system very difficult. Although the API contains most of the desired functionality, these capabilities cannot be utilized in an autonomous agent deployment model.

```
char msg[20];
int myrank, tag = 99;

MPI_Status status;
MPI_Comm_rank ( MPI_COMM_WORLD, &myrank); /* find my rank */

if (myrank == 0) {
    strcpy(msg, "Hello World");
    MPI_Send(msg, strlen (msg) + 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```

Figure 41. Example MPI code (SOHL[+]96).

4.2.4 Message Oriented Middleware.

4.2.4.1 JMS and JMQ. The Java Message Service (JMS) provides a common framework for Java applications to interact with an enterprise messaging system (Sun99c). Enterprise messaging products, commonly called message oriented middleware (MOM), are offered by a variety of vendors (Figure 42). JMS offers a common set of interfaces and semantics in order to access these systems.
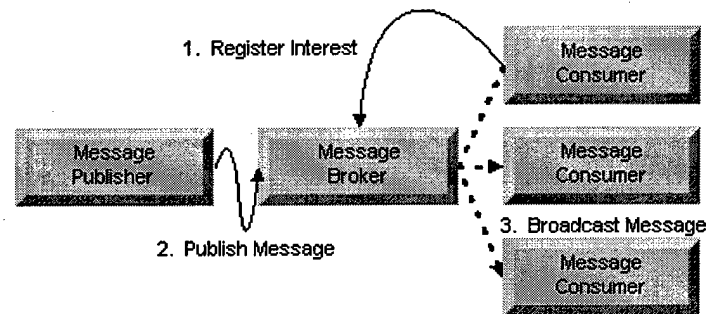
Figure 42.   Message oriented middleware operation (Som97).

The Java Message Queue (JMQ) provides a MOM solution that is JMS compliant (Sun99a). It allows clients to exchange data without the need to utilize low-level communication constructs. JMQ facilitates message passing between different threads within the same process, different processes within the same computer, and separate processes on distributed computers (Sun99b). It accomplishes this by utilizing a communication protocol, a router process to transfer messages, and an API library to interface with the Queue (such as JMS). Therefore, JMS and JMQ must work together to provide a complete messaging solution; JMS for message construction and JMQ for delivery.

JMS messages contain a common header field, a properties section, and the body. The header is made up of 10 sub-fields such as source, destination, time stamp, message ID (sequence number), etc. This obviously represents considerable overhead and most of these fields are unnecessary for this prototype. The properties section allows for the addition of user-specified header fields. Finally, the body contains the actual data to be sent. This can be any serializable object. The rest of the JMS API consists of methods for

100

interacting with a MOM solution. JMS supports calls for all of the desired features, such as multicasting, subscription, security, and event driven operation. However, the actual implementation of these features is dependent upon the host MOM. JMS's power lies in its high level messaging abstraction and its ability to operate with various MOM solutions. Its major weakness is poor efficiency due to large numbers of unused message fields.

The JMQ offers the implementation of the desired messaging features. It utilizes a data centered approach, as opposed to an address centered approach. Applications indicate the type of data they are sending and consequently clients subscribe to the types of data they wish to receive (Figure 43). This provides the capability for multichannel, one-to-one, or one-to-many messaging. All of these constructs work within an event-driven model to provide asynchronous communication. Applications must specify an event handler to respond to specific JMQ events, such as message recieved.
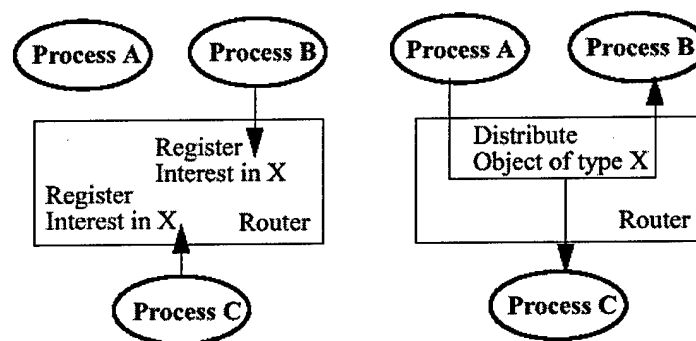
Figure 43.   Java Message Queue subscribe operation (Sun99b).

The JMS/JMQ combination offers a very complete and robust solution for enterprise messaging. But, for this application, the wide array of capabilities is not required and represents unnecessary overhead (Table 15). The JMS/JMQ solution is also deceptively simple in the literature. Message passing appears to be as easy as calling a send routine, yet very few details on how to fully implement a complete solution are given. Additionally, JMQ is only in a version 1.0 beta release. This raises questions as to its reliability, stability, and compliance with stated specifications. In general, JMS/JMQ offers too much capability and intimates a corresponding low-efficiency.

101

Table 15.   JMS/JMQ communications library capabilities.

Requirements	JMS/JMQ
Efficient - low overhead	
1-to-1	√
1-to-many	√
Asynchronous	√
Event Driven	√
Multichannel	√
Subscription	√
High Level Abstraction	√
Flat Learning Curve	
Security Layer	√

4.2.4.2 AgentMOM.   AgentMOM is a communications framework developed by the AFIT Agent Research Group (ARG) (DeL99b). It is designed to explicitly implement the communications required in MaSE designed systems. The hope is that agentMOM will be integrated with the ARG's agentTool for the automatic generation of agent systems. Although agentMOM is termed as a message oriented middleware for agents, it is actually devoid of middleware services commonly associated with MOMs, such as automatic message routing, or queuing. Therefore, a more correct name would be the MaSE agent communication environment (MACE?).

Agents utilizing this framework implement two components, the message handler and the conversation (Figure 44). Agent communication occurs via conversations, as in MaSE. When an agent wants to collaborate, it begins a conversation as a separate thread. The initial message is sent across a socket connection to the recipient's message handler. The message handler monitors a local port for incoming messages, which it passes on to an agent's *receiveMessage* method. The *receiveMessage* routine processes that message and, if appropriate, begins the other side of the conversation in a separate thread. After that initial contact, the conversation is handled by the two conversation threads. Utilizing threads for conversations eliminates agent busy waiting during blocking communication calls.

Messages in this framework are sent as the content in peer-to-peer conversations. AgentMOM does not directly support one-to-many, multicast, messaging. This would

102
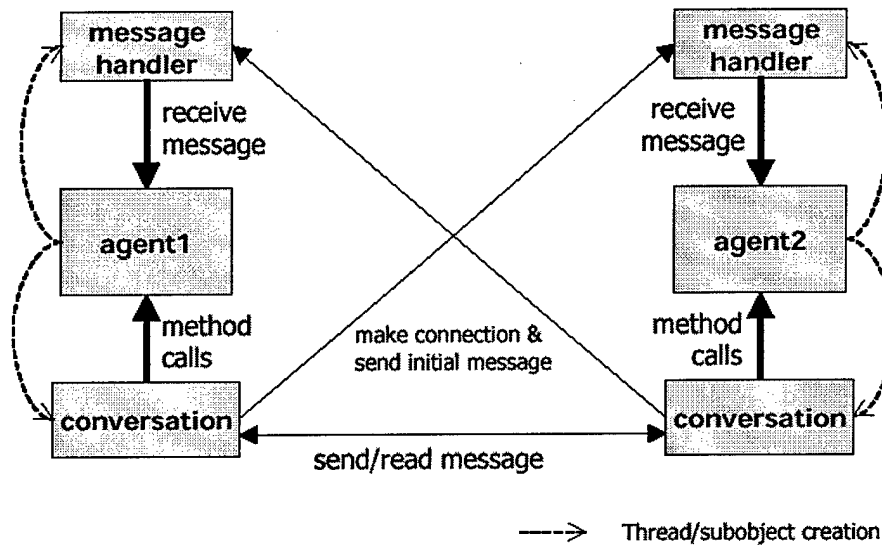
Figure 44. AgentMOM operation (DeL99b).

have to be simulated by using multiple one-to-one calls. The agentMOM architecture uti-
lizes asynchronous, event driven messages and multichannel messaging is accommodated
via multiple conversations all running as separate threads. AgentMOM does not use sub-
scription based channels, instead conversations are initiated and torn down as required.
This would potentially be more efficient if conversations are few and far between, as is the
case with those initiated on virus detection. This lack of a subscription service also alludes
to agentMOM's low level of abstraction. AgentMOM requires the programmer to specify
socket addresses and ports. However, because operations are at this level, performance
gains can be realized through tailoring of the operations to the exact problem domain.
Additionally, this facilitates the addition of extra functionality. For instance, at this level,
security is not implicitly offered, however, the socket constructor could easily be replaced
by one that implements SSL.

AgentMOM offers a medium level abstraction for agent communication. Instead of
middleware services, as the name implies, the library provides base classes and function-
ality to the individual agents. AgentMOM partially defines the structure of the agents
themselves, not just the communications mechanisms. For instance, the passing of re-
ceived messages to an agent's *receiveMessage* method is specified. An additional benefit of

the lower abstraction is performance improvements gained by a reduced number of object layers as well as the capability for implementation tailoring.

This framework appears to be a good solution. It meets all the desired requirements except multicast, subscription, and a high level abstraction (Table 16). But, this can be overcome with a slight performance penalty by making multiple, individual sends. Another problem is the lack of a registry for subscription capabilities. The individual agents must know the location of the other agents in order to utilize their services. In essance, each agent must maintain a registry of all its peers. This is not a big problem in the CVIS architecture, as the location of a Detector's Monitor server should be known, and not needing to run a registry, or utilize its services, may provide performance increases. Overall, agentMOM represents a trade between ease of use and better performance. With that in mind, it meets most of the needs of the system, while simultaneously providing the possibility of good performance, and as a side benefit, implementation details for all the agents.

Table 16. AgentMOM communications library capabilities.

Requirements	agentMOM
Efficient - low overhead	With Tailoring
1-to-1	√
1-to-many	
Asynchronous	√
Event Driven	√
Multichannel	√
Subscription	
High Level Abstraction	
Flat Learning Curve	√
Security Layer	√

4.2.5 Distributed Objects.   The idea of many autonomous objects, each with differing functionality, residing in separate address spaces, and communicating with each other through message passing, fits well within the distributed computing model. The object-oriented nature of this environment gives rise to the distributed-object model (RLC$^+$99). There are several environments available for implementing distributed object computing,

including the component object request broker architecture (CORBA), the distributed component object model (DCOM), and Java remote method invocation (RMI).

CORBA provides an extensive array of services for a distributed-object architecture. The advantage of CORBA is its platform neutrality. Its disadvantage is the very steep learning curve associated with understanding all the available services. This complexity was deemed unnecessary for this initial prototype.

Both DCOM and RMI offer similar capabilities, but each has advantages and disadvantages. RMI is directly tied to the Java language. It uses the Java core classes and methods to perform tasks. The subs and skeletons used to do the RPC object marshaling are generated automatically by RMIC, the RMI compiler. By using core Java, the programmer gains platform independence.

COM/DCOM is language independent, but platform dependent. COM objects can be created, or used, by J++, Visual C++, and Visual Basic. However, COM is interwoven into the Microsoft Win32 environment. The COM API is part of the Windows API. COM objects require the use of the Windows Registry for their brokering, although, Microsoft provides COM/DCOM support for other platforms including Tru64 Unix, OpenVMS, and Solaris. Additionally, Saga Software[1] offers DCOM solutions for many UNIX and mainframe platforms with its EntireX product. So, even though DCOM is tied to Windows, the capability for interoperability with many different platforms does exist.

Java 1.1 provided little security outside of Java's "sandbox" restrictions. However, security has been increased in Java 2 with the addition of a security manager, which must be initialized prior to using RMI. DCOM, being closely linked to NT, uses NT's security services to allow specific rights to objects and users though the use of access control lists. This is all very flexible and easily configurable through the DCOM configuration tool, *DCOMCNFG*.

Java requires a running RMI registry in order to provide object brokering. Server objects are bound or registered with it using a single function call. Clients connect to servers by making requests through the registry. In COM, each object and interface requires

---

[1]www.sagasoftware.com

a unique 128 bit global unique identifier (GUID). This number not only must be unique, but also must be in a specific format. *GUIGEN* and *UUIDGEN* are provided to create these and help ensure uniqueness. These IDs are used as handles to call COM objects and interfaces. The IDs are also used to bind COM objects with the Windows Registry. Objects can be registered/unregistered using function calls, or with the *REGSVR32*. J++ COM wrapped objects can be registered with *JAVAREG*.

With RMI, all object administration must be explicitly programmed into the application code, including access rights and remote functionality. User permissions must be set up correctly at the operating system level. DCOM however has the DCOM configuration tool *DCOMCNFG*. This provides a very easy point and click interface to control security, allowed object functionality, even the location of execution. This is what makes COM, DCOM.

Because most viruses attack the Windows platform, tying the system to DCOM is not a distinct disadvantage. The availability of third party DCOM support for other platforms will enable system expansion to heterogeneous operating system environments. Also, the DCOM security model offers an advantage to the system. These security features facilitate protecting the CVIS from spoofing or viral attacks. These advantages, combined with DCOM's offering of similar services as RMI and CORBA, make DCOM a good distributed computing object architecture for implementing the CVIS. All three methodologies offer similar performance (MCD99).

The two major weaknesses with DCOM for this project are a lack of persistence and the need for callbacks. In COM/DCOM, all of the objects are stateless (MCD99). They have a lifetime which is limited to their usefulness. On instantiation, a remote COM object is created. It is then used until the calling program releases it. This does not meet the requirements of an autonomous agent or system component persistence. If Detectors are continually created and destroyed, all of their associated antibodies need to be recreated and all the memory cells are lost. The system needs a way to maintain states. CORBA on the other hand offers persistence through unique object handles that allow connection to specific object instances.

106

The other problem is the need for object callbacks. Detectors need to be instantiated with a pointer back to their Monitor. This way, they are able to send a warning message if a virus is found. The problem lies in how other agents communicate when no pointers are available, such as inter-Monitor warning messages. The DCOM RPC methodology is not well suited for the web of communication channels required in this system. These problems and limitations with DCOM make it unsuitable for an agent based computer virus immune system.

Although messy for system message passing, remote objects could be used effectively in the Helper, Classifier, and Cleaner chain. Each of these agents are used very rarely. When a viral detection event occurs, each of these is instantiated and is required for only a short period. In this case, the COM lack of persistence is actually an asset. They can be created when needed and removed after their tasks are performed. These agents should be remotely executable so that they can be placed locally with the infected file.

The use of distributed objects and RPCs offers a very high level of abstraction to network communication, but does not meet many of the other ideal system requirements (Table 17). Each of the three methodologies offers similar capabilities, but with substantially different learning curves and security features. The lack of persistence and the poor performance of these architectures make them largely unsuitable for an agent-based application.

Table 17.   Distributed object communications capabilities.

Requirements	CORBA	DCOM	RMI
Efficient - low overhead			
1-to-1	√	√	√
1-to-many			
Asynchronous			
Event Driven	√	√	√
Multichannel	Multiple RPCs	Multiple RPCs	Multiple RPCs
Subscription			
High Level Abstraction	√	√	√
Flat Learning Curve	High	Medium	Low
Security Layer		√	Minimal

4.2.6 Agent Development Kit. There exists a wide variety of libraries, frameworks, or agent development kits (ADKs) for use in developing agent-based systems (Ret99). In this section, Aglets and Voyager are investigated for use in the CVIS. These were chosen from the myriad of available specimens based on their maturity and the availability of adequate documentation of their features and capabilities.

Most of the developed ADKs focus on the use of mobile agents to implement system functionality. One of the main tenets of parallel computing is the importance of minimizing communication costs. With mobile agents, the entire agent program, data, functions, and all, must be packaged up and transported. Messages on the other hand, which only pass the required data, represent a much lower communication cost. In the CVIS, there is no advantage to mobile agents which would justify their high communications cost. Additionally, the security problems are large and many, which no current ADKs completely address (KT98) (See Section 2.4.2 for a more complete discussion of mobile agent security and performance limitations). Therefore, although both Aglets and Voyager support mobility, only their messaging features are evaluated for this project.

4.2.6.1 Voyager. The Voyager ADK provides interagent communication through RPCs. In this way, only synchronous messaging can occur. Voyager agents can get around this limitation by using non-blocking function calls (Mah00). The non-blocking calls provide a placeholder for a function's return value. This placeholder can be queried at a later time to determine if the data is available and, if so, the data can be read.

Voyager also supports "multicast" function calls on a group of agents. Agents can subscribe to, or become members of, spaces. A function can be called on all members of the space by calling *Multicast.invoke* on the space. The space then passes this call to all of its member agents that support the calling interface.

The use of the Voyager API is very similar to distributed object paradigms, such as RMI. The Voyager server must be run on each host, similar to the RMI registry. Calls are then made to the server to instantiate remote agents and then interact with them. This presents the same problems with persistence seen in RMI and DCOM. However, Voyager does provide an advanced activation capability, by which calls to objects that have been

terminated may complete successfully. Through the use of activation proxies, programs may survive restarts and continue operation without interrupting client requests (Mah00).

Voyager does provide a security layer, but it simply restricts agent functionality according to whether an agent is local or remote. For instance, remote objects are not allowed to delete files on a local machine. Beyond this, any form of authentication or encryption would have to be added by the programmer.

Voyager provides all of the capabilities of the distributed object framework, with the addition of groups and multicast features (Table 18). A slightly higher level of abstraction makes Voyager a little bit easier to use than RMI and much simpler than either DCOM or CORBA. However, it suffers from many of the same problems as these systems, due to its reliance on RPCs. The greater capability of Voyager also comes at a very high cost. Remote method calls involving three arguments are twice as slow using Voyager compared to RMI. Additionally, Voyager is greater than three times slower than socket messaging (HYI98). Because of its capabilities, Voyager would be a better choice than some of the other distributed object architectures, but because of its high overhead and the limits of RPCs, Voyager is unsuitable for this application.

Table 18.   Voyager communications capabilities.

Requirements	Voyager
Efficient - low overhead	
1-to-1	√
1-to-many	√
Asynchronous	Future Reply
Event Driven	√
Multichannel	√
Subscription	√
High Level Abstraction	√
Flat Learning Curve	Medium
Security Layer	Remote or Local Agents

4.2.6.2 Aglets.   The Aglets agent system does not permit agents to invoke each other's methods (KT98). This enforces autonomy in agent behaviors. In order to per-

form agent collaboration, message passing is used.[2] However, only synchronous, one-way, and future reply operations are supported. Aglets do not support multicast or subscription capabilities.

Aglets do support an event driven model, although this is not in support of messaging. A key feature of Aglets is this event model. Specific methods are called during the Aglet "lifecycle." For instance, two of the methods are *onCreation* and *onArrival*. These functions are executed when an Aglet is instantiated and after it arrives at a new host location.

Aglets must exist in a certain location known as a context (Smi99). Within the context, Aglets interact through proxies. It is through the proxies that messaging occurs. These operations are very straight forward; Aglets provide the intuitive *sendMessage* routine. Replies can be returned by simply calling the *sendReply* method directly on the recently received message (Figure 45).
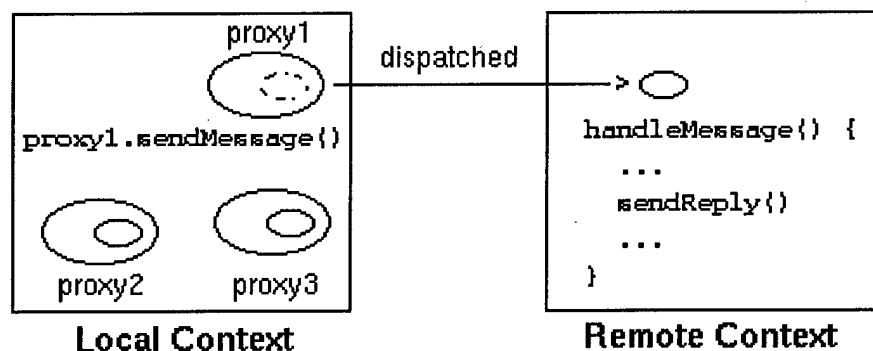
Figure 45.   Aglet message passing operation (Smi99).

The Aglets ADK provides a message class for the construction of messages. Each message contains a string attribute, which indicates the agent "kind." An agent's *handleMessage* method can then perform specific operations by distinguishing between message kinds. Messages can also contain key and value pairs. These are used to hold message data. A key's value may be any Java object. This use of string keys represents overhead that must be passed around with each message.

---

[2]Although, this is implemented as calls to an agent's message handling methods, such as *handleMessage*.

110

The Aglets package meets most of the ideal communication package requirements for the system (Table 19). It provides a simple API for messaging, but provides less capability than Voyager. The ease of use in the Aglets package does not make up for a projected poor performance or a lack of support for a few of the necessary system capabilities.

Table 19.   Aglets communications capabilities.

Requirements	Aglets
Efficient - low overhead	
1-to-1	√
1-to-many	
Asynchronous	Future Reply
Event Driven	√
Multichannel	√
Subscription	
High Level Abstraction	√
Flat Learning Curve	Medium
Security Layer	Remote or Local Agents

4.2.7 Conclusion.   For this study, many different solutions for agent collaboration were investigated. Candidates from different architectures including shared memory, message passing, message oriented middleware, distributed objects, and agent development kits were reviewed. The capabilities of the ten candidates were compared against a set of ideal system capabilities required to support the immune system model, the anti-virus problem domain, and distributed autonomous agents. A key concern is communications efficiency, which is an enabling attribute for problem decomposition, agent collaboration, and system scaleability. A summary of the investigation is shown in Tables 20 and 21.

The worst library for this application is MPI. Although it provides highly optimized communication between processes, it is specifically tailored for symmetric parallel applications. It has been used very successfully in parallel applications for scientific computing, but its constructs do not fit the client/server environment.

Next, are the distributed object methodologies. All three of these utilize procedure calls on remotely instantiated classes. This requires some form of object broker to connect clients to servers. Running and using this service induces extra system overhead and a

Table 20. Communications library capabilities summary part I.

Requirements	JSDT	JavaSpaces	agentMOM	JMS/JMQ	MPI
Efficient - low overhead	√		By Tailoring		√
1-to-1	√	√	√	√	√
1-to-many	√	√		√	√
Asynchronous	√	√	√	√	√
Event Driven	√	√	√	√	
Multichannel	√	√	√	√	√
Subscription	√	√		√	Groups
High Level Abstraction	√	√		√	
Flat Learning Curve	√	√	√		
Security Layer	√		√	√	

Table 21. Communications library capabilities summary part II.

Requirements	CORBA	DCOM	RMI	Voyager	Aglets
Efficient - low overhead					
1-to-1	√	√	√	√	√
1-to-many				√	
Asynchronous				Future Reply	Future Reply
Event Driven	√	√	√	√	√
Multichannel	Multiple RPCs	Multiple RPCs	Multiple RPCs	√	√
Subscription				√	
High Level Abstraction	√	√	√	√	√
Flat Learning Curve	High	Medium	Low	Medium	Medium
Security Layer		√	Minimal	Remote or Local	Remote or Local

possible single point of failure. With this architecture, the biggest problem is collaboration between two or more already operating agents. Only CORBA can support the persistence of the remote object. In general, RPCs are layered on top of lower level networking functions and induce overhead in their attempt to make the function call appear to be operating locally. Because the methods on peer agents are directly manipulated, this architecture does not support the autonomous agent paradigms very well.

The agent development kits directly support multiagent system construction. The basic services and classes of both Voyager and Aglets facilitate ease in agent construc-

tion. However, these systems emphasize the functionality of agent mobility above agent performance. This emphasis of functionality over performance is also evident in the Java Messaging Service. Like Voyager and Aglets, the extensive feature set of JMS is not required for this application, so its inherent overhead for providing those services make this methodology unsuitable.

The final candidate on the cut list is JavaSpaces. This architecture is unique in that it uses shared memory "spaces" in order to pass data between agents. The problem with JavaSpaces is their efficiency. In order to read or remove an entity from a space, a template of the desired object is presented to the space for comparison. Only an object that matches this template is returned. The need to perform this matching operation on every read from the space creates considerable overhead above and beyond the simple networking required to deliver the object to and from the space. This leaves the Java Shared Data Toolkit and agentMOM as the only two feasible solutions.

AgentMOM captures the idea of agent conversations and integrates this functionality into individual agents very well. Unfortunately, its name is somewhat deceiving as no messaging middleware is provided. Instead, agentMOM contains classes to implement multithreaded socket communications. All agents must know the address and port of the destination agent and explicitly call TCP/IP sockets to perform communication. This does not provide the one-to-many messaging, subscription naming services, or a high level of abstraction desired for this system. However, the JSDT does provide all of the desired system requirements.

Although the Java Shared Data Toolkit provides all necessary capabilities, it does not integrate well into the agent conversation paradigm in its suggested implementation. Additionally, the JSDT multicast operation introduces problems for the anti-virus domain. In order to set up multicast, agents subscribe to a channel and then receive all messages placed on that channel. This is useful in the CVIS for operations such as dispensing vaccinations to all the Detector agents. However, like many security directives, such as AFCERT warnings, the Controller and Manager agents need to ensure compliance with the vaccination. This requires a conversation, including verification receipts, with every agent. A multithreaded set of individual conversations with all channel subscribers can

113

more effectively support vaccination replies. Therefore, what is really needed is a combination between the agentMOM multithreaded conversation model with the high level JSDT communications library constructs.

4.2.8 CVIS Communications Design.   The JSDT constructs are combined with the agentMOM architecture to provide a hierarchical communications network that supports the system, network, and local CVIS levels (Section 3.4.3). By utilizing the JSDT session constructs, the implementation can create multiple sessions to logically isolate conversations at the appropriate level (Figure 46).
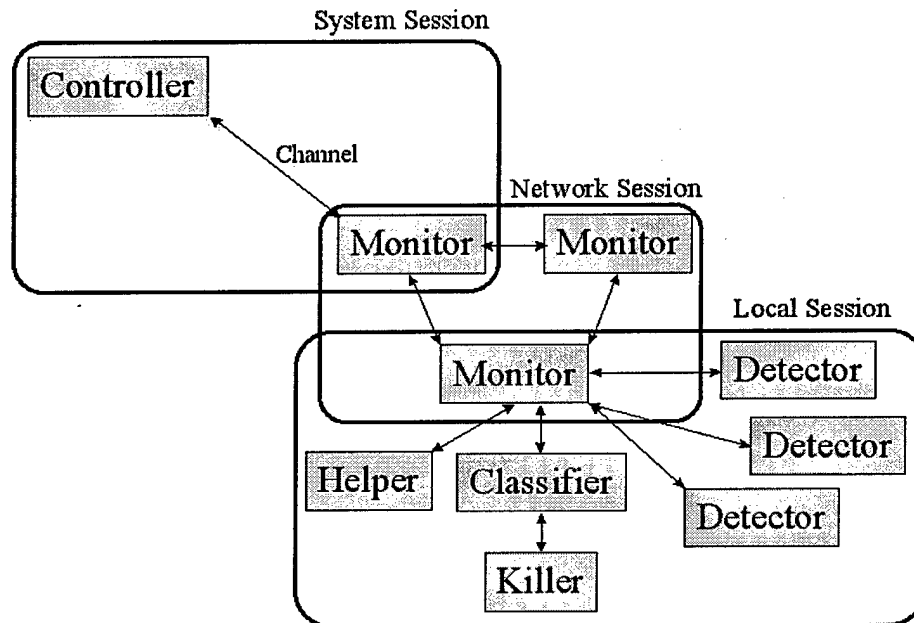
Figure 46.   Session level logical view.

The system level session encompasses the regional (or global) Controller agents and their assigned network Monitors. At the network level, various sessions connect local Monitors in order to pass on epidemic warning messages. Finally, many local sessions connect the Monitors to their assigned agents. Within these sessions, multiple channels are created in order to carry on interagent communications (Figure 47). Each of these conversations is implemented as a separate thread.

114
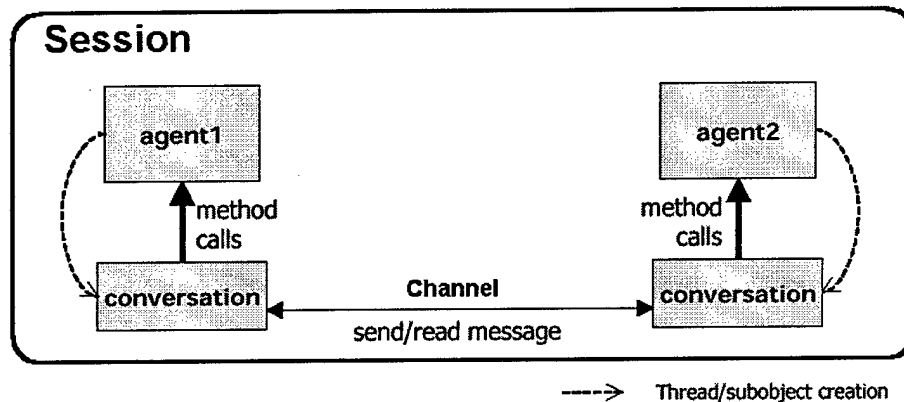
Figure 47.   Channel level logical view.

The operation of conversations are based on agentMOM constructs. However, because JSDT channels are used, the message handler is unnecessary and agents can use the session members list as a naming service to invite other agents to join a conversation. Conversations are implemented as separate threads in order to allow multiple concurrent conversations within a single agent. This also enables agents to better enforce accountability in multicast type conversations. The following is the agent conversation process.

1. The agent joins a session.

2. The agent looks up the other agents in their shared session to begin a conversation with one (or more) of them.

3. The agent creates a new conversation thread.

4. The conversation creates a new managed channel within the session.

5. The conversation invites the other agent to join the channel.

6. The agents converse by passing messages back and forth.

7. The initiating conversation thread expels the other agent from the channel.

8. The conversation thread closes the channel.

By utilizing the capabilities of the JSDT, combined with the overall architecture of agentMOM, a solution that meets all of the system requirements is obtained. Furthermore, this implementation elegantly captures the agent conversation paradigm.

115

4.3 Summary

The complete system design is presented in this chapter. It follows the MaSE software engineering methodology in order to natively integrate agent concepts into the design process. A distributed agent computer virus immune system is designed that follows the system model (Section 3.4.3, Figure 28) for agent deployment and responsibility assignment. The local model (Section 3.4.4, Figure 29) for fighting viruses is encapsulated within the Detector agent and its associated antibodies. However, full system effectiveness is only accomplished through interaction among all the agents. This communication occurs in an architecture modified from agentMOM (Section 4.2.4.2) and implemented within the JSDT (Section 4.2.3.1) provided constructs. The message passing design allows for a hierarchical communications infrastructure. The results from testing the Java implementation of this design can be found in the following chapter.

V. Experiments, Results, and Analysis

The previous chapter documented the overall system design, agent decomposition, and the virus detection methodology. This chapter presents the system test plan along with the results and analysis gained through system testing. In a broad sense, the tests provide insight into the system efficiency, effectiveness, and scaleability. These broad categories are looked at within two areas, non-self detection and the distributed agent system performance. The detection algorithms are the "heart" of the system. These operate on every local machine within the Detector and Antibody agents. Alternatively, the agent performance area concerns the parallel and distributed aspects of the CVIS. This includes communication performance and the ability of the multiagent system to coordinate actions effectively. Well coordinated agents are important, but if non-self cannot be found, the system as a whole is of course ineffective.

5.1 Test Plan

A sound experimental design involves several key factors. Theses include deciding what questions are to be answered by the experiment, identifying variables that are influential in determining code performance, and collecting a set of applicable test problems (CDM79). The complete test environment also needs to be specified in order to ensure repeatability. The set of experiments outlined in this test plan provide quantitative and statistical data in order to provide a basis for qualitative analysis. The analysis of the data gained from the test plan begins in Section 5.2. The CVIS test plan begins with the experimental questions.

5.1.1 Experimental Objectives. The purpose of the system experiments is to understand the performance implications of the CVIS agent components and their communications. The prototype only has a limited fielding of agents, but a total system would include a hierarchy of many, possibly geographically separated, agents types. The objectives of system experimentation are to gain insight into the efficiency, effectiveness, and scaleability of an actual fielded system. These objectives lead to the following test questions, which are expanded into test metrics:

Efficiency

- What network loading does the system induce?

- What is the system execution time?

- What is the system scan time or scanning rate?

- What is the system communication time to computation time ratio?

- What are the system memory requirements?

- What is the system response time to epidemics?

Effectiveness

- Is the system capable of determining self from non-self?

- Is the system capable of detecting known and unknown non-self?

- Is the system adaptable to a changing definition of self as programs are added?

- Are successful detectors (antibodies) maintained?

- Are distributed agents and the JSDT a viable architecture for a CVIS?

- Does distributed detection improve the system through collective self-defense?

Scaleability

- What are the effects of adding additional agents?

- Do system resource limitations impact scaleability?

5.1.2 Influential Variables. The influential variables are those items that may be controlled, or uncontrolled, which have an affect on system performance (CDM79). Each of these involve engineering tradeoffs in system design and they may or may not be independent. The variables and some of their affects are listed below:

Warning threshold – What level of detection is required to raise an alarm? Can affect Type I and Type II error rates.

Update time – How often are ineffective antibodies regenerated? Need to balance exploration and exploitation of the antibody landscape.

Antibody length – How many bytes are there in an antibody string? Affects memory usage and antibody effectiveness.

Number of antibodies in a detector – How many antibody strings are in each detector? Affects the probability of detection.

Number and types of agents – How many agents and what types are fielded? Affects almost every aspect of efficiency, effectiveness, and scaleability.

Agent locations – Where are the agents fielded? Affects communications.

File contents in self and non-self – What bit patterns make up self and non-self? Self = non-self can lead to an autoimmune reaction.

Length of self and non-self – How large is self and the total file system? Affects scan time and negative selection time.

Frequency of change in self – How often are programs and data added to the system? The CVIS may not be able to quickly adapt to a rapidly changing notion of self due to the negative selection time.

Known virus database size – How many identification strings are stored for classification? Affects the classifier identification time and storage space.

Disk I/O speed – How fast can files be read? Affects the speed of scanning, negative selection, and classification.

5.1.3 Measures of Performance. Measures of performance (MOP) are a crucial factor in a computational experiment (CDM79). They are numeric indicators that give insight into system efficiency, effectiveness, and scaleability.

M1 – What is the generated antibody diversity (Euclidean distance)?

M2 – How do/does the number of antibodies, antibody length (byte), file system size (MB), and the threshold level affect the false positive (Type I) error rate?

M3 – How do/does the number of antibodies, antibody length (byte), file system size (MB), and the threshold level affect the false negative (Type II) error rate?

M4 – How do/does the number of antibodies, antibody length (byte), and file system size (MB) affect the scan time?

M5 – How do/does the number of antibodies, antibody length (byte), and size of self (MB) affect the negative selection time?

M6 – What is the communications library efficiency (communications time)?

M7 – How many simultaneous conversations can one agent support (CPU loading)?

M8 – How does the number of agents affect the network loading (communications time)?

M9 – How quickly are epidemics suppressed?

M10 – How many antibody candidates are required to generate the required number of antibodies?

5.1.4 Test Inputs.   There are three basic sources of test problems, those that arise naturally in practice, ones that are specially constructed to test a particular aspect of the code, and finally, randomly generated problems (CDM79). Additionally, it is desired to test this new application against a common industry benchmark. This test plan uses all four.

The test problems for a CVIS are sets of self and non-self strings. In order to test the operation of the antibodies, some are assigned to predetermined values. To test the CVIS's ability to function in a large search space, randomly generated sequences are used. Finally, the system is tested against actual user programs and viruses in order to understand the system's applicability to the real world problem domain. The complete set of test inputs can be seen in Table 22.

Of particular use for the real world problem set is the European Institute for Computer Anti-Virus Research (EICAR) standard anti-virus test file (Duc99). This file contains of a set of 68 bytes of ASCII printable characters:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTI-VIRUS-TEST-FILE!$H+H*
```

Table 22.   Test inputs.

Number	Description	Purpose
1	Self - All 1's Non-self - All 0's	Does negative selection, detection, costimulation, and memory work?
2	Self - All 1's Non-self - Random characters	Does the CVIS detect a diverse threat?
3	Self - same as Non-self	Does the CVIS suppress autoimmunity?
4	Self - Randomly generated Non-self - Randomly generated	Does CVIS operate in a large search space?
5	Self - Application program suite Non-self - EICAR test program	Does CVIS operate correctly in a real world environment?
6	Self - Application program suite Non-self - TIMID virus	Comparison of this CVIS to other research.

The purpose of the file is to provide a safe target for testing the operation of anti-virus software. The file is easy to use and non-infecting. It is an executable COM file that only prints the message *EICAR-STANDARD-ANTI-VIRUS-TEST-FILE!*. Most commercial anti-virus software products have scan strings that recognize the EICAR test pattern.

One of the key reasons for utilizing an immune system model of operation is to recognize as of yet unknown viruses. Therefore, a modified version of the EICAR test string is used as a new, unknown "virus." For this purpose, EICAR was modified so that it now prints *Paul Harmer-s test Virus XxXxXxXx!!* instead. This new non-infecting strain of EICAR goes undetected by Norton AntiVirus.

Testing the system against a common industry benchmark is desired in order to compare the efficiency and effectiveness of the proposed CVIS against other solutions. Unfortunately, there does not exist such a baseline. This prototype represents one of the first CVISs constructed. However, tests were performed in (FAPC94) in order to validate their r-contiguous bits theoretical derivations against actual data. For these tests, the TIMID virus (Lud96) was used to infect COM files. The algorithm was run against a varying number of antibody strings in order to discover the probability of failure, $P_f = 1 - P_{detection}$, and the initial number of immature antibodies required to create a desired number of naive strings after negative selection. These same tests are run against the

121

agent-based CVIS in order to compare it with the UNM work and to test the system using an actual file infector virus, TIMID.

TIMID is a simple file infecting virus (Lud96). It only infects one file on each execution. Its targets are COM files residing only in TIMID's local directory. It does not hop across directory structures. Additionally, TIMID has the nice feature of outputting the name of its victim.

TIMID is an appending file infector that adds 5 bytes to the top of a file and an additional 300 to the end. No stealth capabilities are employed, so victim files sizes can be seen to grow by 305 bytes, along with an appropriate file date alteration. All these features make TIMID an excellent test subject because it can be controlled and its effects are known. Furthermore, it is a commonly known virus that can be detected and removed by all current AV suites. TIMID, EICAR, and the generated problem sets become inputs to the test cases.

5.1.5 Test Cases. The test cases are designed to gather data for every MOP. In each test case, either an influential variable is changed, or a static system property is measured to understand system performance. Each test case may cover more than one MOP and it may also gather more than one data point. The test cases are enumerated in Table 23.

Each test is run multiple times in order to account for statistical variations in system performance. From this data, an average, a high, a low, and the variance are computed. These are used to understand the statistical significance of the system performance measurements.

5.1.6 Testing Platform. The CVIS is tested on the AFIT Bimodal Cluster (ABC) pile of PC's. This is a heterogeneous system consisting of 22 variously configured Pentium II and Pentium III CPUs connected by a fat tree Gigabit and 100baseT switched Ethernet backbone (Figure 48). The ABC is under constant development with a rapidly changing system configuration. However, the complete system depicted in Figure 48 is expected to be available for CVIS testing. Each machine within the cluster is dual bootable as a

Table 23. Test cases.

Number	MOP	Input	Measurements/Desired Output	Variable
1		1	Is the system capable of determining self from non-self?	
2		2	Can a diverse threat be detected?	
3		3	Does the CVIS suppress autoimmunity?	
4		5	Can known and unknown viruses be detected?	EICAR vs. EICARnew
5	M1	4	Distance between detector antibodies	Generation method
6	M2, M3	4	Type I and Type II error rate vs. number of antibodies	Number of antibodies
7	M2, M3	4	Type I and Type II error rate vs. antibody length	Antibody length
8	M2, M3	4	Type I and Type II error rate vs. file system size	File system size
9	M2, M3	4	Type I and Type II error rate vs. threshold level	Threshold level
10	M4	4	Scan time vs. Number of antibodies	Number of antibodies
11	M4	4	Scan time vs. antibody length	Antibody length
12	M4	4	Scan time vs. file system size	File system size
13	M4	4	Negative selection time vs. number of antibodies	Number of antibodies
14	M4	4	Negative selection time vs. antibody length	Antibody length
15	M4	4	Negative selection time vs. size of self	Size of self
16	M7		Processor loading vs. number of simultaneous agent connections	Number of agents
17	M8	1	Communications time vs. number of simultaneous agent connections	Number of agents
18[1]	M9	1	Epidemic elimination time	
19	M10	6	Number of immature antibodies required to generate a number of antibodies	Number of antibodies

1. Test 18 was not completed due to time constraints.

Windows NT or Red Hat Linux system. All systems are booted as NT systems to reflect the most common virus target platform and current Air Force server standards.
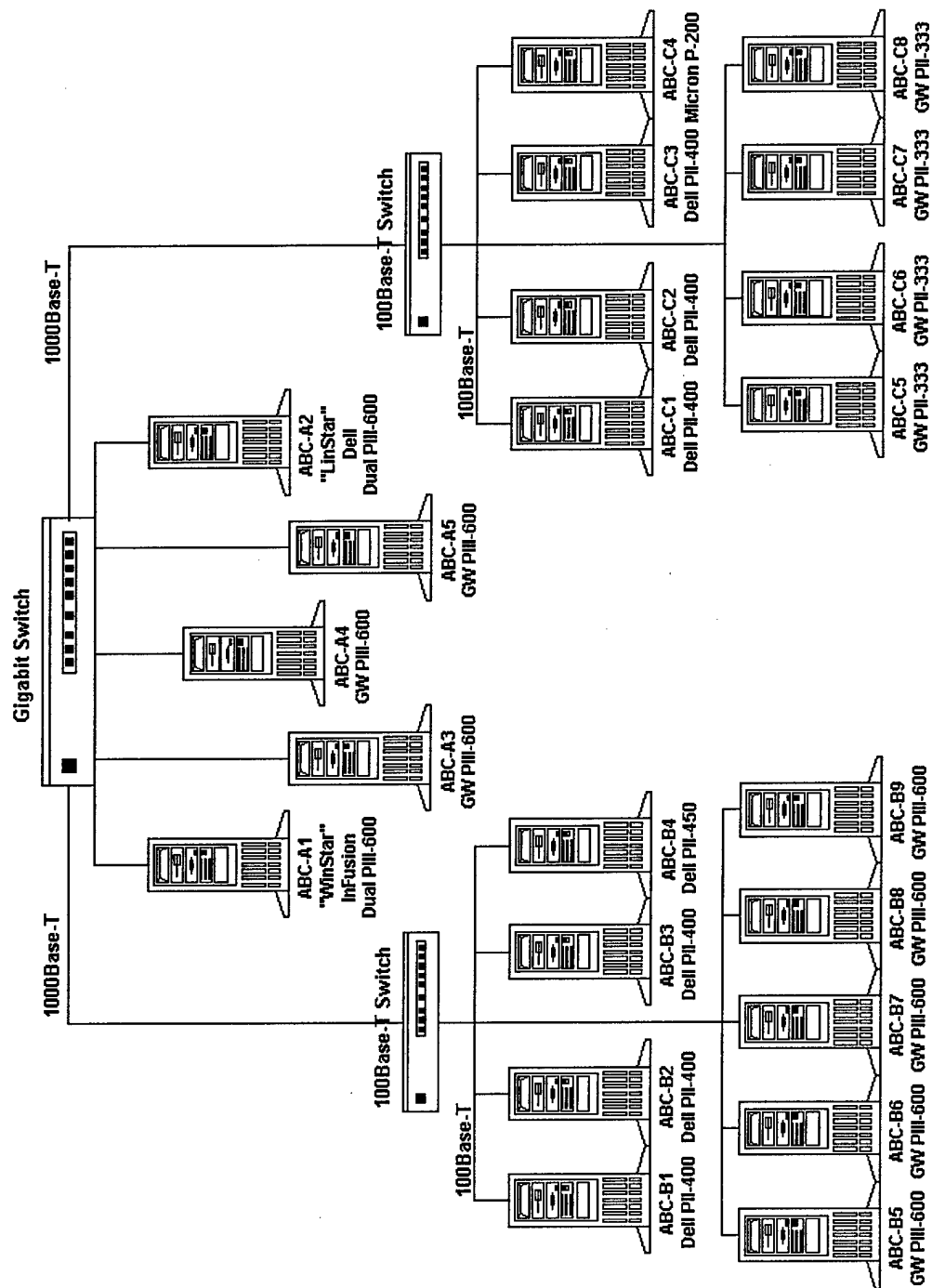
Figure 48.   AFIT Bimodal Cluster physical architecture (Current as of Mar 00).

The ABC is a closed environment that is representative of a PC LAN network, the target implementation platform of the CVIS. Increasing the test platform to include Win95, Linux, or even Solaris machines would be a good test suite for understanding the performance of the CVIS in the enterprise environment.

For this project, the complete agent deployment utilizes 13 machines, as depicted in the physical system deployment diagram (Figure 38). CPUs and their associated agents are added to the system up to the complete deployment to test the system scaleability. The configurations of the machines are shown in Tables 24 and 25.

Table 24.   Pile of PCs machine configurations part I.

Feature	ABC-A1	ABC-A3	ABC-A4	ABC-A5	ABC-B5	ABC-B6
CPU	Dual PIII 600	PIII 600	PIII 600	PIII 600	PIII 600	PIII 600
Cache	512K	512K	512K	512K	512K	512K
RAM	256MB	384MB	384MB	384MB	384MB	384MB
HD	25.5GB	6.5GB	6.5GB	6.5GB	6.5GB	6.5GB
OS	1	2	2	2	2	2

1. Windows2000 Advanced Server
2. Windows2000, RedHat Linux 6.1
3. WindowsNT 4.0 Workstation, RedHat Linux 5.2

Table 25.   Pile of PCs machine configurations part II.

Feature	ABC-B7	ABC-B8	ABC-B9	ABC-C1	ABC-C2	ABC-C3	ABC-C4
CPU	PIII 600	PIII 600	PIII 600	PII 400	PII 400	PII 450	PII 400
Cache	512K	512K	512K	512K	512K	512K	512K
RAM	384MB	384MB	384MB	128MB	128MB	128MB	128MB
HD	6.5GB	6.5GB	6.5GB	8.4GB	8.4GB	6.9Gb	8.4GB
OS	2	2	2	3	3	3	3

5.1.7 Compiler and Runtime Environment.   All code is developed using IBM's VisualAge for Java version 2.0 (see Section 3.5 for the Java language selection study). There are no compiler optimization settings. The Java JDK v1.1.8 is used as the Java Virtual Machine (JVM) runtime environment. This version utilizes Symantec's just in time (JIT) bytecode compiler as the default in order to improve runtime performance. There are many options for JVMs and compilers, but it is not the purpose of this research

to evaluate different Java platforms. Although not contained in this test plan, alternate Java runtime environments could be used to possibly enhance system performance. Several different JIT compilers are available, such as IBM's Jikes compiler. Another option, since platform independence is not a goal, would be to use a native code compiler, such as the one available from the GNU organization. These are not considered in this prototype implementation, but could be considered in future research.

5.2 Virus Detection

The ability to effectively detect non-self is the primary goal of this system. But, detection must also be efficient and scaleable. If the negative selection and scan algorithms are not efficient, their execution times become too long to be practical or unobtrusive to the user. Additionally, if execution times increase radically with the addition of new (self) software, or the addition of new antibodies through vaccination, then the scaleability of the system limits its long term use. This section examines the effects of the number of antibodies, antibody length, detection threshold, and the size of the file system on the efficiency, effectiveness, and scaleability of the detector/antibody agents. These areas are explored through the antibody diversity, negative selection time, scan time, matching function probability density, and the system error rate measures of performance (Section 5.1.3).

5.2.1 Antibody Diversity. The purpose of this artificial immune system is to search through the set of finite length strings, X, in order to find elements of non-self (Section 3.2). By definition, because $X = S \cup N$ and $S \cap N = \emptyset$, those strings that are not part of self, S, are elements of non-self, N.

The process of negative selection is a search through X given an input self set S. In this prototype, candidate solutions to this search are generated pseudo-randomly at antibody creation. The search space is large and exact areas of self and non-self are unknown. Therefore, the goal is to generate a diverse set of antibodies in order to explore as large an area of the space as possible.

As a measure of diversity, Euclidean distance is used. Therefore, this test requires generating $O(N^2)$ distance measures. For this test, twenty antibodies, 4 bytes long are

generated. With this many antibodies, 400 distances are generated. For N much greater than this, the test quickly becomes unwieldy. The calculated distances are normalized based on the maximum possible distance, D_{max}.

$$X \in \{0 \ldots 255\}^l$$

$$l = 4$$

$$A, B \in N$$

$$\forall A, \forall B \; D_{A,B} = \sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2 + (A_3 - B_3)^2 + (A_4 - B_4)^2}$$

$$D_{max} = 255 \cdot \sqrt{l}$$

For the pseudo-randomly generated set, the average normalized distance between antibodies was 0.38 with a standard deviation of 0.12 (Figure 49). For comparison, Figure 50 is the normalized distance distribution for twenty evenly spaced antibody strings. This set of antibodies produces an average distance of 0.37 with a standard deviation of 0.24. The sawtooth variations in the "exponentially" decreasing trend are due to integer rounding of the byte values.
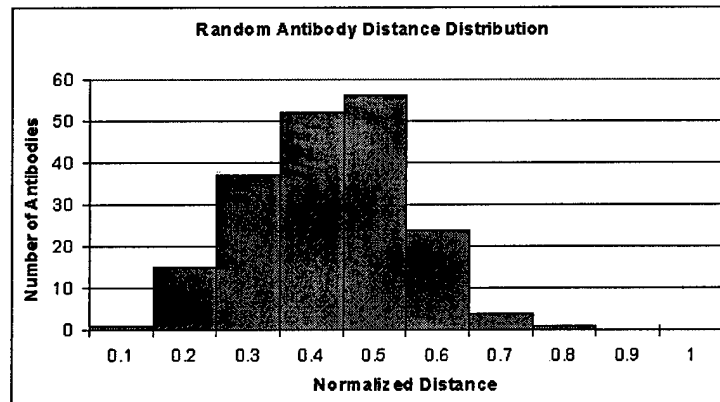

Figure 49.   Antibody diversity for randomly generated detector strings.

Both figures give insight into the diversity of the antibody population, whether randomly created or generated by deterministic means. The random strings are slightly farther apart from each other on average, but the deterministic antibodies have a wider spread. This indicates that the deterministic antibodies cover a more diverse set of the search space, whereas the pseudo-randomly generated strings are clustered together with a few
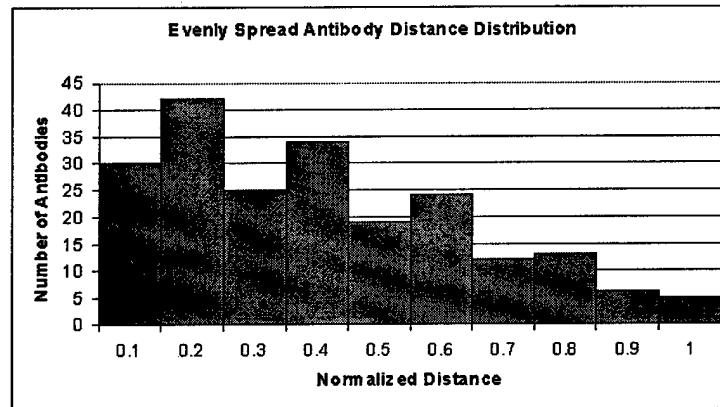
127

Figure 50.   Antibody diversity for evenly spread generated detector strings.

outlying samples. The figures also depict this trend. The 1% difference in the average distances is equivalent to a distance difference of 5.1 in this space. This does not appear to be very significant, but for larger antibody lengths, and therefore larger search spaces, the 1% difference will become more statistically significant.

There is no way to know where in the search space non-self strings are found. They may be evenly dispersed, or found in clusters. For this system, random string creation probably performs as well as any other method. This is a byproduct of the No Free Lunch theorem (WM97). This theorem states that a stochastic search technique may perform well on a specific problem instantiation, but this is no guarantee that this same method will perform well on all instantiations of this problem. In essence, there is no one search technique for all problems that can perform better on average than a random search. The only way to improve the search is to bring in domain knowledge. Because negative selection is used, self must be predefined as an input to the algorithm. If this self were categorized, then the generation of non-self scan strings could be better directed to non-self areas of the search space.

5.2.2 Negative Selection Time.   The negative selection algorithm represents an investment that the system must make in order to remove the possibility of false positive errors (Section 3.2). The current algorithm sequentially checks each antibody against all bytes in the known self space. This requires adding each byte from self to the sliding

window and then comparing each antibody bit by bit. Negative selection is performed after each antibody string is randomly generated. If a match on self occurs, the antibody is regenerated and re-tested from the beginning of self. Theoretically, the negative selection time should grow linearly with respect to the number of antibodies, the length of each antibody, and the size of known self. This is because each byte in each antibody must be checked against every self byte.

$Number\ of\ antibodies = N$

$Antibody\ length\ (bytes) = L$

$Size\ of\ self\ (bytes) = S$

$Sliding\ window\ shuffle = O(L)$

$Bit\ compare = O(8L)$

$Negative\ selection = O(N(S(L + 8L)))$

$= O(NSL)$

The experimental results accurately follow the expected theory as the time tends to double as the number of antibodies are doubled (Figure 51). These tests were accomplished using 1K of randomly generated self bytes. The results also depict larger variations in the negative selection time occurring with the smaller antibody sets. With the larger antibody sets, the effect of an additional scan on the total time is minimal. However, with only two antibodies, the occurrence of a single self match results in a negative selection time that is half again the length of a no-match scan. The variations in the average negative selection times, and therefore the expected linear growth rate, are due to the occurrence of self matches along with the cost of regenerating and re-censoring the replacement strings.

Figure 52 depicts the effects of antibody length on the negative selection time. At lengths greater than 2 bytes, the negative selection time grows, although this growth is almost imperceptible. The change in length also induces very little variance in the negative selection time. However, a 2 byte string produces very dramatic increases in the censoring time with accompanying wide variance between runs. This is widely divergent from the theory. The observed 25 fold increase in the negative selection time is due to the increasing
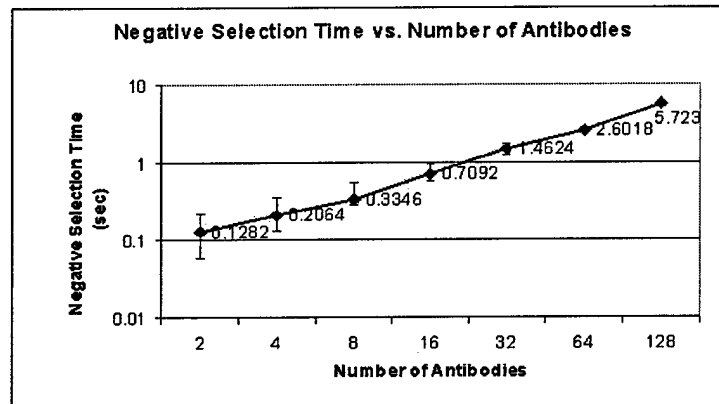
129

Figure 51.   Negative selection time versus the number of antibodies per detector.

generality of a 2 byte string; an increase in detector length produces an antibody that is increasingly tuned to a more specific virus. Somewhere between 2 and 4 bytes there is a sensitivity point, before which a very large number of matches on self occurs. The result is a much larger negative selection time in order to find the required number of 2 byte combinations that do not match self. The 4 byte antibody falls at the beginning of this trend. It has a negative selection time that is slightly greater than the 8 byte case, which is also divergent from the expected theory.
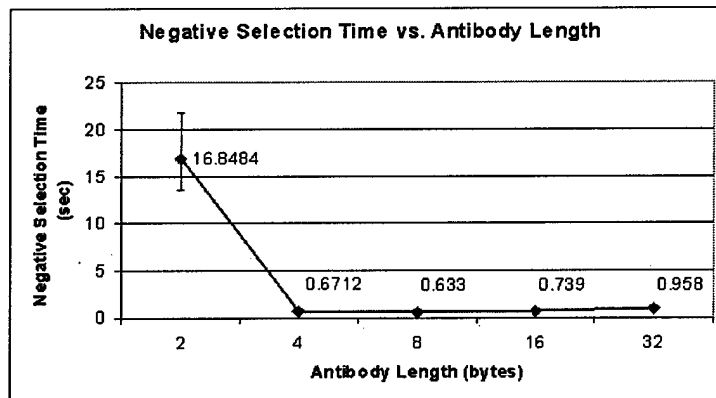

Figure 52.   Negative selection time versus antibody length.

The previous tests utilized 1K of self bytes in order to censor the antibodies. Figure 53 shows the effect of the length of self on the negative selection time. The linear increase validates the theoretical results and the data also shows very little variation in the

experimental times, which results in imperceptible error bars in the figure. The standard deviations are 4% or less of the average value. The system produces 16, 8 byte antibodies against 1MB of known self in approximately 10 minutes. Using the more general, 2 byte antibodies could cost over 25 times that on average.
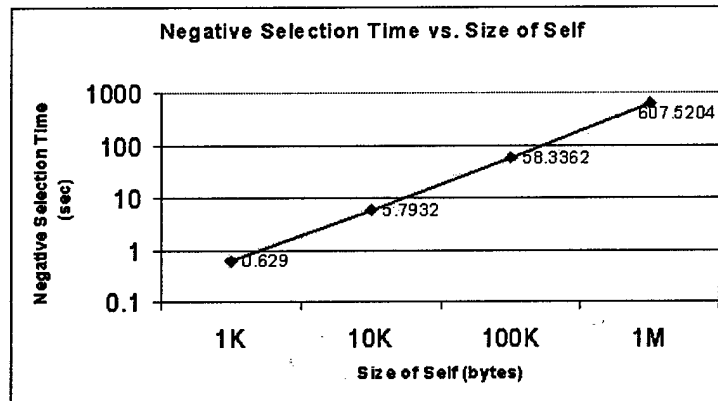

Figure 53.   Negative selection time versus the size of self.

The generation of correctly censored antibodies produces the core components for virus detection by an artificial immune system. File systems are large and growing. For example, an 8GB hard drive is now considered on the small side for commercially produced PCs. The current performance of this system would produce 128, 4 byte antibodies against 8GB of self in 1.45 years!

$Negative\ Selection\ Time = \frac{5.723 secs}{1KB\ self} \cdot \frac{1 \times 10^6 KB}{1GB} \cdot 8GB = 45,784,000\ sec$

$= 1.45\ years$

Clearly, this is too long to be practical and any reduction in antibody length or detection threshold would only increase this time. Algorithmic and implementation improvements are required to reduce the negative selection time to a usable duration.

5.2.3   Scan Time.   The scan operation represents the heart of an anti-virus system. Ideally, this algorithm should run as quickly as possible in order to be unobtrusive to the user. Although even current systems often fall short of this goal.

131

Theoretically, the scan time of this system is directly proportional to the amount of data being scanned, the number of antibodies, and the antibody length. The scanning algorithm must read in each byte of the file system, add it to the sliding window, and then compare the window against the antibody string bit-by-bit.

$Number\ of\ antibodies = N$

$Antibody\ length(bytes) = L$

$Size\ of\ file\ system(bytes) = X$

$Sliding\ window\ shuffle = O(L)$

$Bit\ compare = O(8L)$

$Negative\ selection = O(X(L + N(8L)))$

$= O(NXL)$

Experimental results hold true to theoretical expectations. The number of antibodies in a detector directly affects the scan time (Figure 54). As the number of antibodies is doubled, the scan time is also doubled. Compared to the negative selection algorithm (Section 5.2.2, Figure 51), scanning produces negligible variations in execution time. During scanning, if a match occurs, the offending file and its bound antibody are added to a list. This requires no regeneration or re-scanning as in negative selection. Therefore, scan times are almost constant between runs.
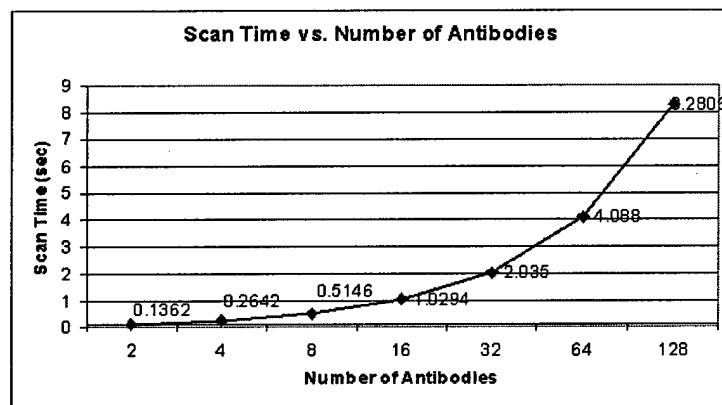

Figure 54. Scan time versus the number of antibodies per detector.

132

Antibody length also affects scan time linearly, however there is a very small multiplier reducing the effect (Figure 55). The time only increases about 3% with the addition of an extra byte to the string. This results in a 6% jump when doubling the antibody length. In general, these results give a few specific, long, antibodies an advantage over many short strings. Longer strings can be used with only small performance ramifications. The long antibodies also result in relatively short negative selection times due to their specificity (Section 5.2.2, Figure 52). The trade off is in the ability to effectively search the larger space created by utilizing specialized detectors.

Scan Time vs. Antibody Length
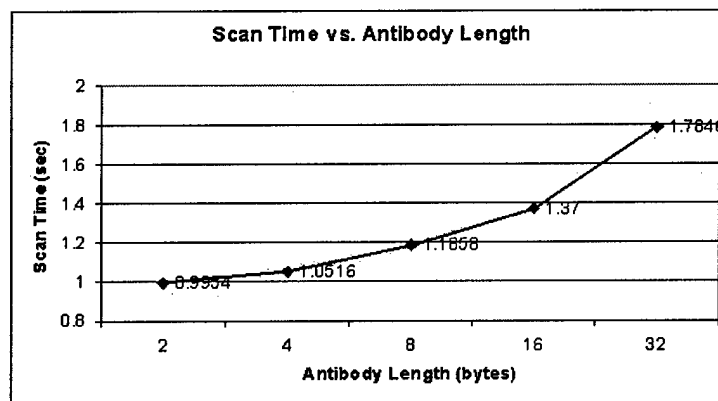
Figure 55.   Scan time versus antibody length.

The experimental results also parallel theory with respect to file system growth (Figure 56). Increasing the file system size 10 fold also increases the scan time by a factor of 10. The 2MB file system is scanned in $19\frac{1}{2}$ minutes. Extrapolating out, an 8GB file system with 128, 4 byte antibodies would take 1.05 years to scan.

$$Scan\ Time = \frac{8.2866 secs}{2KB\ self} \cdot \frac{1 \times 10^6 KB}{1GB} \cdot 8GB = 33,146,400\ sec$$

$$= 1.05\ years$$

Scanning is faster than negative selection because files only need to be opened once and compared against all antibodies (Section 5.2.2). The current negative selection algorithm requires opening every file in the system once for each antibody. Therefore, the I/O system overhead is incurred multiple times unnecessarily. This observation gives insight into possible algorithmic improvements for negative selection. As with negative selection,

Figure 56. Scan time versus file system size.

the scan time is too long to be of practical use. Algorithmic and implementation improvements are needed to make the system usable.
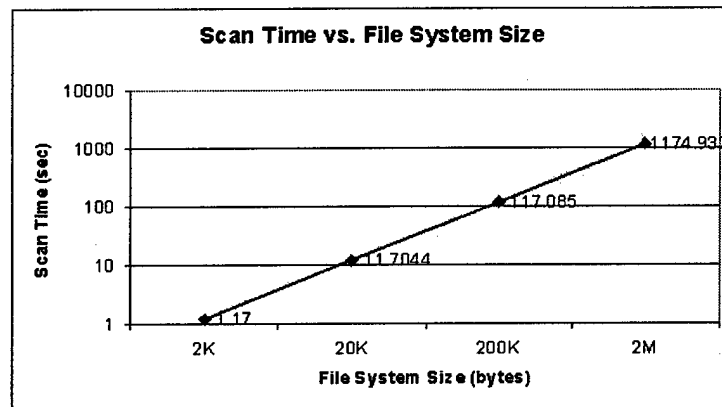
5.2.4 Matching Function Value Density. The matching function value probability density plot produces a pictorial representation of the antibody specificity. It also allows the system designer to match the antibody, matching function pair with an appropriate detection threshold.

Figure 57 depicts the probability density histograms for 8 byte antibodies tested against various sizes of self. Each of these graphs are identical and also match the 8 byte antibody graph in Figure 58. From these experiments, it is obvious that file size has no affect on antibody specificity. Even the very small variations are not greatly effected by an increasing number of samples. The 8 byte antibody, combined with the Rogers and Tanimoto matching function results in a stable value probability density.

While the size of the scan space has no effect on antibody specificity, string length has a significant effect (Figure 58). As the antibody length increases, the probability distribution is compressed towards the 30%-40% band. This reflects the increasing specificity of the longer antibody strings. Also, variance in band probabilities decreases with antibody length, although the variability in all the probability values is very low. The 4 byte antibody experimental results compare almost exactly with those gained in the matching rule selection tests (Section 3.3, Figure 25). This similarity includes the operational problem
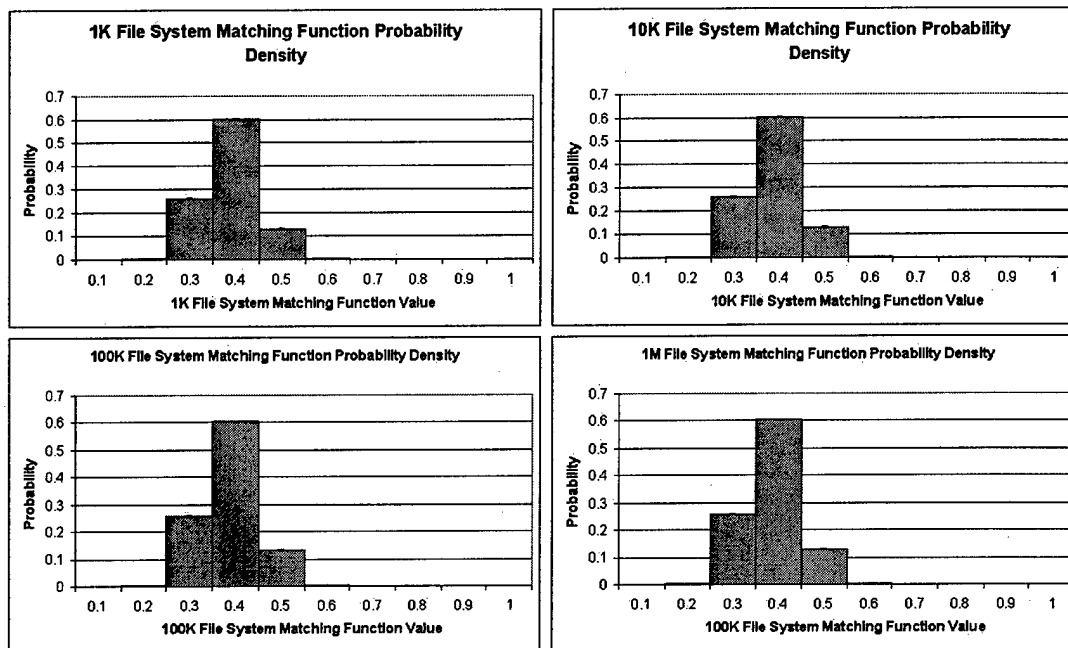
134

Figure 57. Matching function probability density versus file system size.

of the wide gap between a perfect match and the first function values appearing between 60%-70%. The gap problem is somewhat alleviated by using 2 byte antibodies (Figure 59). The 2 byte antibody probability more closely matches the ideal case (Section 3.3, Figure 23). This indicates that very general scan strings may provide better detection results, but these small strings induce other problems.

The antibody length should be selected based on the desired specificity of the antibody search strings. While 2 byte strings more closely reflect the ideal case, their extreme generality results in a high negative selection cost. Additionally, the antibody strings are (most often) used to find viral machine code instructions. On Intel hardware[1], 2 bytes only code for a few simple instructions, such as a stack pop to a register (Table 26) (TS91). Increasing the string length to 4 bytes allows the antibody to cover almost the entire basic instruction set, except for direct memory addressing instructions. Therefore, utilizing antibodies of length 4 or greater is more appropriate considering the target platform.

---

[1]DOS-based systems operating on Intel microprocessors are the host to the greatest number of viruses (Section 2.1.1).
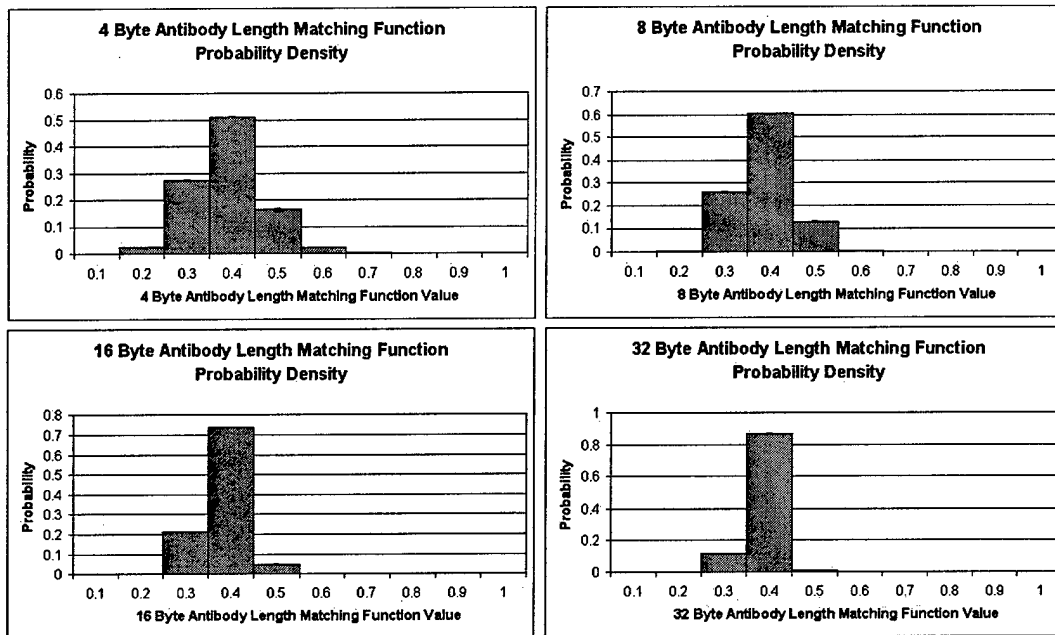
Figure 58.   Matching function probability density versus antibody length.

Table 26.   Intel 8088 processor instruction lengths (TS91).

Type	Name	Mnemonic	Length
Data Transfer	Pop to Register	POP	1
	Push to Register	PUSH	1
Arithmetic	Increment Register	INC	1
	Compare Immediate	CMP	2
Control	Jump on Greater	JG	2
	Loop	LOOP	2

5.2.5   Error Rate.   The system's error rates reflect its ability to detect self and non-self appropriately (Section 3.2). The false positive rate should always be zero. This is ensured in advance by the negative selection algorithm. By initially censoring strings against self, no future self matches should occur. The false negative rate should also ideally be zero. Any percentage higher than this indicates the system's relative inability to detect the presence of non-self. This rate can fluctuate dramatically because of the stochastic nature of the problem. The antibody strings are randomly generated, as are the appearance of viral infections. So, the system parameters, such as antibody length, the

136

Figure 59.   Matching function probability density for two byte antibodies.

number of antibodies, and the detection threshold must all be tuned in order to minimize the false negative rate.

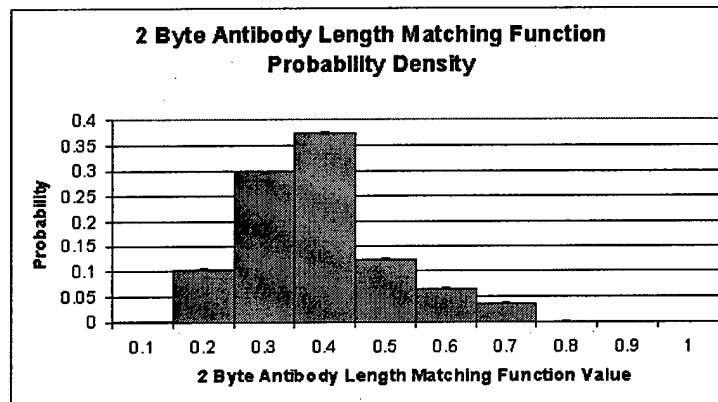Increasing the number of antibodies linearly decreases the average false negative rate (Figure 60). This test utilizes 4 byte antibodies and a 0.7 detection threshold against 1K of randomly generated non-self bytes. The error bars indicate the maximum and minimum values to understand the complete range of values. Due to the probabilistic nature of the problem, even 64 antibodies can fail to find non-self the same as a single antibody. Conversely, the best run of 64 or greater number of antibody strings found all the non-self files. In order to generate a consistently low error rate, 128 or more antibodies are required per detector. However, this results in higher negative selection and scan times. A trade off between speed and coverage must be made. This design trade is the idea behind fast scanning Complement agents (Section 4.1.1). These agents were not included in the prototype implementation (Section 4.1.2), but their use could be explored in future versions.

As previously shown, the length of the antibody affects its specificity as a detector (Section 5.2.4). The error rate for long detectors should be greater than the smaller, more general strings. This is a by product of a long detector's need to search an exponentially larger space in order to find a match. This trend is evident in Figure 61. At lengths greater than 4 bytes, the antibodies have a complete inability to find 1K of random non-self bytes. The extremely general, 2 byte strings are able to find all non-self files with no

137
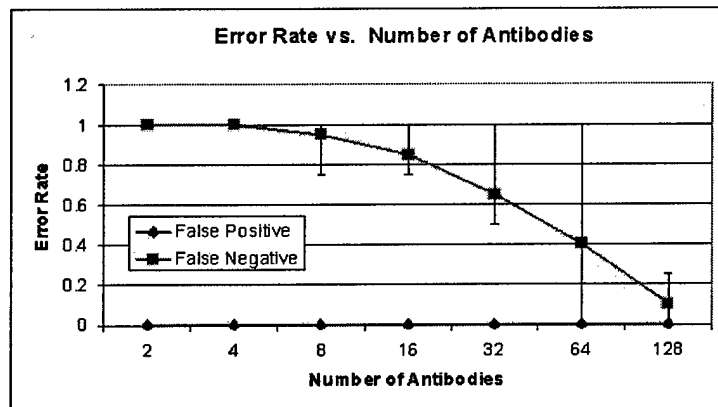
Figure 60.   Error rates versus the number of antibodies per detector.

variance. In the middle between these two extremes is the 4 byte antibody. On average, these perform better than the longer strings, but they also only detect non-self 20% of the time. The variance seen with this length is indicative of its position between too general and too specific. The random generation of 4 byte antibodies can place them on either side of a present or future non-self boundary. For these tests, a 0.7 detection threshold is used. The error rate graph (Figure 61) is a reflection of the probability of a matching value occurring about this threshold (Figures 58 and 59). Therefore, selecting the antibody length determines its specificity, but this must be matched with an appropriate activation threshold in order to obtain the desired error rate. In essence, reducing the detection threshold creates a more general detector, no matter what its length.

Figure 61.   Error rates versus antibody length.

138

Figure 62 depicts the affect of the detection threshold on the false negative rate. For this test, each run consists of a detector with 32, 8 byte antibodies. The 100% false negative rate at a 0.7 threshold matches the same result in Figure 61. These results are also in line with the 8 byte matching function probability density (Figure 58). At threshold values less than 0.7, the antibody set becomes an increasingly effective non-self discriminator. An activation threshold of 0.55 results in a 100% effective detector with no variance.
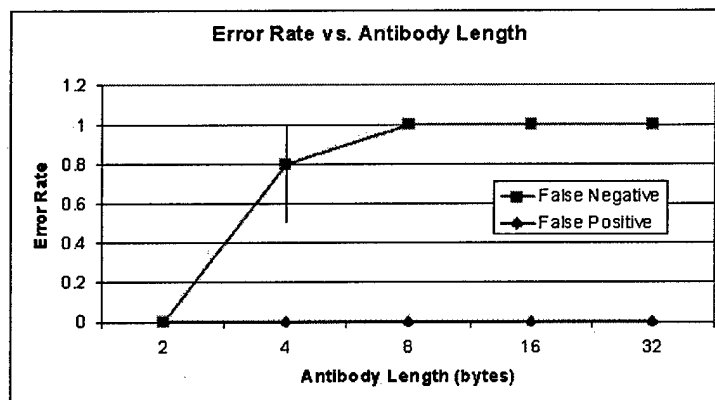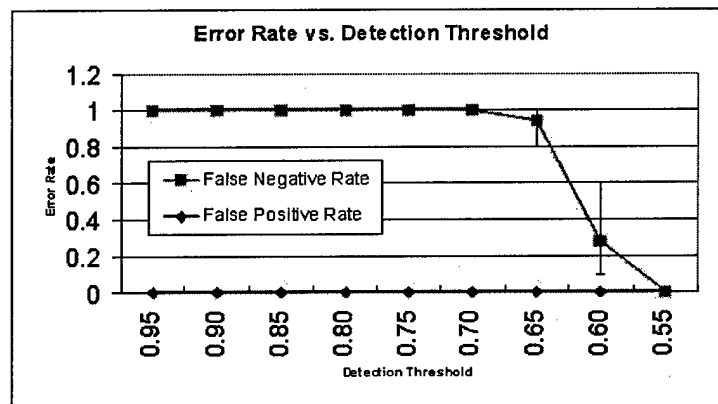

Figure 62.   Error rates versus detection threshold.

The final test examines the affects of the file system size on the error rate (Figure 63). The results indicate only a minor influence, with wide variations. This is not surprising as a larger set of non-self bytes simply gives the detector more chances to encounter a match. For these tests, detectors utilizing 16, 8 byte antibodies searched up to a 2MB file system containing up to 1MB of random non-self bytes. In practice, the likelihood of 1MB of non-self appearing on an individual system is all but impossible. Because most viruses are smaller than 5KB (Section 2.1.4, Table 2), the accumulation of 1M of non-self bytes would require a significant number of simultaneous infections. This is an event this is so remote that its occurrence is impossible without sabotage. Therefore, this example is mostly pedagogical. In a practical environment (5K-10K bytes of non-self) the size of non-self has no measurable effect on the false negative error rate, except to keep it high.

These results indicate that each detector should field as many generic antibodies as possible in order to minimize the false negative rate. However, the use of highly generic, as well as large numbers of antibodies contribute to an increased negative selection time.

139
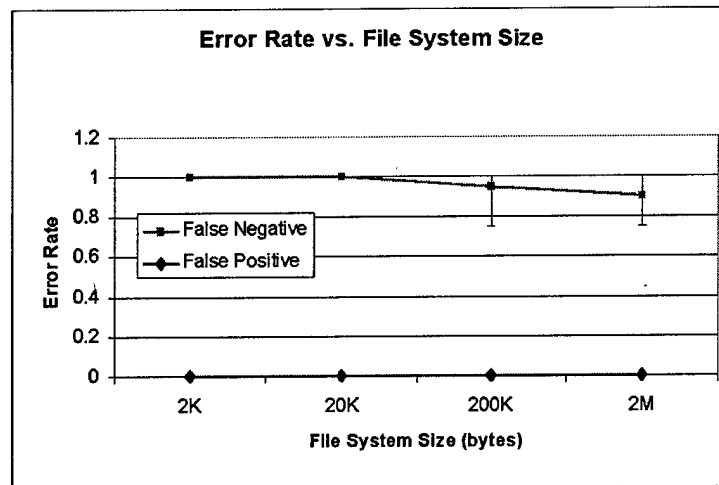
Figure 63.   Error rates versus file system size.

But, negative selection is necessary to force the false positive rate to 0%. An engineering trade-off must be made between negative selection time and system effectiveness. Once the desired antibody length is selected, the detection threshold must be tuned to the antibody matching function probability density in order to create a system that actually detects non-self with the desired frequency.

5.2.6 Real World Effectiveness.   The previous tests have shown that the system operates as designed and is able to successfully detect the existence of non-self within a set of self strings. However, these results were gained by testing the system against randomly generated self and non-self bytes. In order to be truly effective, the system must be able to detect actual malicious code among a larger set of known self applications.

The first test against other than random bytes uses a polar input set. For this test, self is made up of all ones, while non-self consists of all zeros (Section 5.1.4, Table 22). Interestingly, the randomly generated 8 byte antibodies have a harder time finding this consistent non-self set (Figure 64). Full detection only occurs with an activation threshold of 0.4 or less. The 100% error rate difference between 0.45 and 0.4 is indicative of the consistent, polar nature of the self and non-self sets. Once one detector is able to bind with a string of zeros, it is able to bind with all of non-self. This results in a 0% false negative rate once the detection threshold is crossed. This test does not provide much

140

useful information in itself other than the dramatic effect a proper detection threshold selection can make. But, by comparing with the similar data obtained using random non-self bytes (Figure 62) and interesting difference emerges. This test shows 100% detection at a threshold of 0.4, while with random strings, 100% detection occurs at a 0.55 threshold. Previous data indicated that the detection threshold should be tuned based on the antibody length. Additionally, this test indicates that tuning should also be done based on the contents of self and non-self. This data also shows that antibody generation could be improved based on knowledge of existing self and non-self bytes. An *a priori* examination of non-self would have revealed that a single antibody pattern consisting of all zeros could have matched, with the highest affinity possible, all non-self in this system. Because of the influence of search space contents on system effectiveness, tests against actual viruses are conducted.
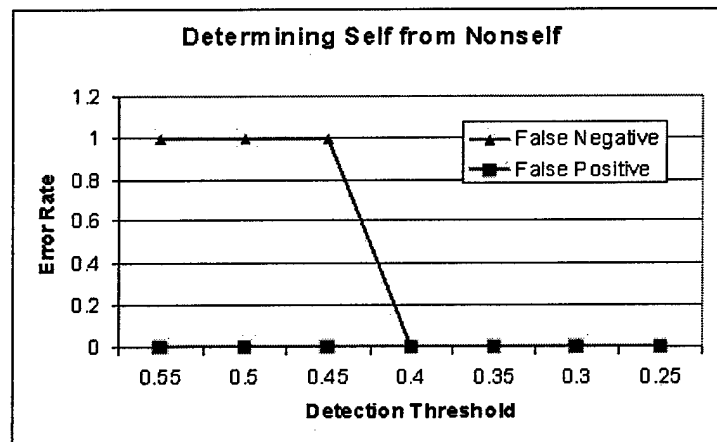
Figure 64.   Detector's ability to detect non-self.

The real world virus test utilizes test input five (Section 5.1.4, Table 22). This test suite consists of 196KB of application programs and 136 bytes in the two EICAR "viruses" (Section 5.1.4). This test also reveals the system's ability to detect, as of yet, unknown viruses. For example, Norton AntiVirus (NAV) can detect the EICAR68 test string 100% of the time, while it has a 100% false negative rate for the newly created EICARPAU test string. Figure 65 presents the error rates for various detection thresholds. The detection rates represent the total rate for multiple runs of variable number of antibodies per detector.

141

A roughly 10% false negative error rate is the best result when using a detection threshold of 0.6 or less.
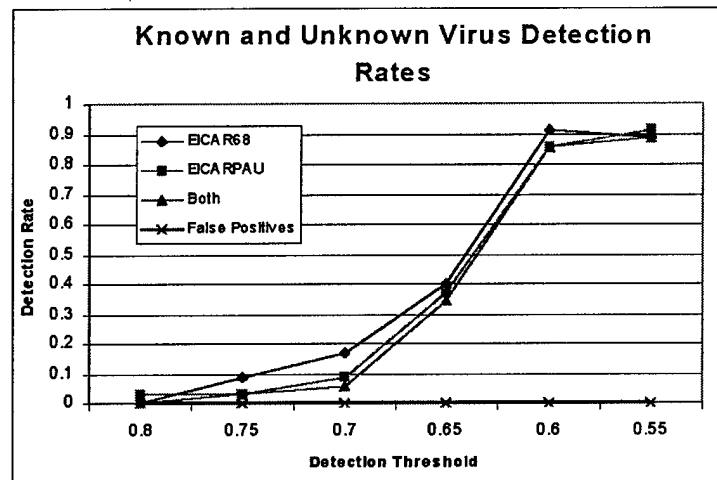
Figure 65. Detection rate for known and unknown viruses.

In general, the system was able to detect the new virus strain at a rate only slightly less than that of the known virus. Additionally, the system found both non-self files at a rate equal to, or slightly less than, the least detected strain. In these cases, the addition of affinity maturation (Section 2.2.3.2) to improve the antibody false negative rate could be highly useful. An affinity maturation capability could either evolve antibodies to recognize one virus very well, or evolve a general detector that binds to both strings equally.

A tuned detection threshold results in an 89% detection rate for both strings. NAV produces a constant 50% false negative rate. This system out performs NAV for detection thresholds of 0.6 or below and indicates that the theoretical basis for a CVIS is sound. The original problem statement (Chapter I) emphasizes the inability of current anti-virus software to adapt and recognize new viruses. This system is able to detect the new strain with approximately a 9% false negative rate. The trade off with the immune system methodology is the probability of detection, while current systems utilize deterministic scanning to give 100% detection of known viruses. However, through careful tuning, a 0% false negative rate can be obtained (Figure 62). Beyond this, additional coverage can be

142

gained by utilizing distributed detectors that share successful antibodies. An improved error rate can be gained in this manner through a multiagent collective self defense.

5.3 Multiagent Operation

5.3.1 Antibody Candidate Pool Size. The experiments on the system negative selection time suggest that a performance increase can be gained by over generating the number of required antibodies and then censoring this large pool down to the required number. However, such an algorithm requires understanding what size the initial pool of uncensored scan strings should be. Dr. Forrest's research on the r-contiguous bits algorithm validates theoretical results that the required number of initial strings, N_{R_0}, grows with the the probability of a match, the number of final strings required, and the size of self (FAPC94). The Rogers and Tanimoto similarity rule produces similar results.

This experiment varies the detection threshold and the number of final antibodies required against the Timid virus infected application suite, input 6 (Section 5.1.4, Table 22). The results indicate that the size of N_{R_0} increases linearly with the number of required antibodies and exponentially with a decrease in the detection threshold (Figure 66). The higher detection thresholds all require the approximately same number of initial candidates, with a break in this trend occurring at a threshold of 0.65. The required number of candidates increases dramatically thereafter. This phenomenon is roughly the inverse of the results seen in Figures 62 and 65. As the detection threshold decreases, the antibodies become more general. This results in an increased number of matches on self during censoring, and improved non-self detection during employment. The small false negative rates that are required for an effective system require the up front investment in a large antibody candidate pool. The near 100% detection rate seen at a threshold of 0.6 requires about 4 times the number of immature strings as naive ones. In order to obtain a 0% false negative rate at a 0.55 threshold, a 23 to 1 ratio is required. This is a reflection of the increased negative selection time vs. coverage tradeoff seen in earlier experiments (Section 5.2.2).

Once again, the performance of the system requires tradeoffs in coverage, speed, and memory. The selection of the system parameters, such as number of antibodies per detector and the detection threshold, can have a dramatic affect on the system efficiency

143
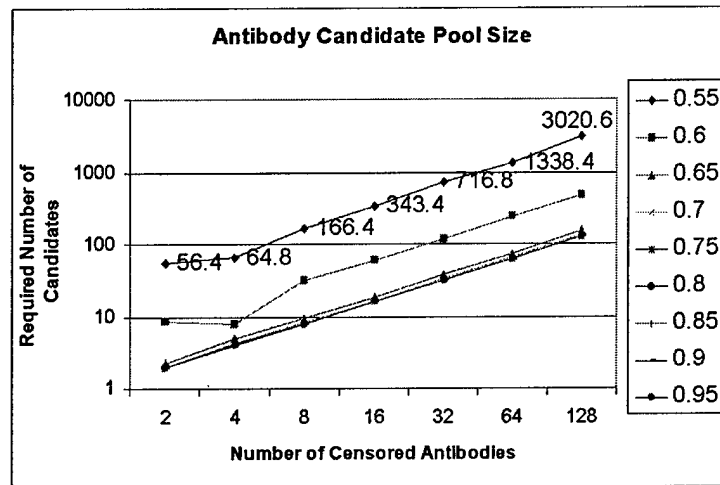
Figure 66. Effects of matching threshold on negative selection candidate pool size.

and effectiveness. At a detection threshold of 0.55, generating 128 antibodies requires an initial pool of 3020 candidates on average. Previous results indicate that at least 64, 4 byte antibodies, at a detection threshold of 0.60 or less is required in order to reduce the false negative error rate to within effective limits. This will require the generation of several hundred to several thousand candidate antibodies for censor.

The UNM research on the use of the r-contiguous bits measure indicates that generating 46 antibodies against 512 bytes of self requires a repertoire of 34,915 candidates on average and results in a 15.7% false negative rate (FAPC94). This agent-based CVIS is able to create 64 antibodies using an average initial repertoire of 1338 candidates and a threshold of 0.55. This results in an average false negative rate of 0% (Figure 62). The system is able to out perform initial r-contiguous bits prototypes and is also more efficient in candidate pool size.

5.3.2 Communications Performance. The Java Shared Data Toolkit (JSDT) is the communication library chosen for this prototype (Section 4.2). JSDT was selected based on its ability to best support the multiagent environment and the logical system hierarchy (Section 3.4.3). The JSDT provides a multithreaded communications backbone with client registry, naming, and lookup services. All messages in JSDT are passed as Data objects. These objects encapsulate the actual data sent by an agent along with other fields,

144

such as the sender's name and the data's priority level. This all represents communications overhead. To understand the effects of this overhead, experiments are performed to gain insight into the role of JSDT communications on the system effectiveness, efficiency, and scaleability.

In order to communicate, all agents must register themselves with a session. The JSDT Registry holds pointers to the sessions and agents subscribed. The Registry may reside on either a local or remote host. Agents joining the system incur an initial communications cost registering with the appropriate session (Section 4.2.8, Figure 46). The results of agents subscribing to local and remote Registries can be seen in Figure 67. The slightly higher average cost of a local registration is counter intuitive, as is the wide variation in connect times. All observed registration times were either 0msec or 10msec for both the local and remote connections. The local test had one 10msec run more than the remote experiments, which leads to the difference in the average values. A larger sample set would probably generate equal average connect times for both cases. But, the area of interest is the polar nature of the connect times. All JSDT communication calls are asynchronously carried out by spawning new threads. It is hypothesized that the polar registration times are the result of chance in the system's entry point into the thread execution queue. The longer, 10msec times are the result of waiting for the round robin queue to come around and schedule the registration thread.
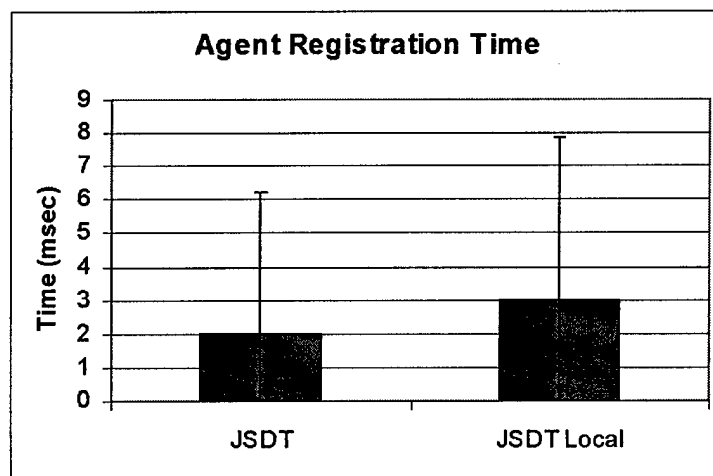
Figure 67.   Agent registration communication time.

145

After registration, agents use the registry in order to lookup peers for communications. This results in a remote agent connection cost. Hirano, Yasu, and Igarashi performed a comparative study of several Java distributed object communications methodologies including RMI, HORB CORBA, Voyager agents, and sockets (HYI98). Tests are performed using JSDT and related to the data presented in (HYI98). JSDT performs remote agent connection by requesting the list of agents registered with a session. What is returned is a list of agent names. These strings must then be parsed for the correct conversation partner. This method of operation results in agent location transparency along with the flexibility of operating on ASCII strings. However, the system incurs a large cost acquiring and sending the array of string names (Figure 68). The cost of a local name lookup is 0 due to the high speed of memory copy versus the low speed network connection to a remote host. In general, JSDT connects faster to a remote agent than the Voyager ADK, but half again slower than RMI, or HORB CORBA. DCOM was not tested due to its inability to connect to existing remote objects (HYI98). After obtaining a reference to a remote agent, data must be sent to it for processing. This is commonly accomplished via a remote method call.

Figure 68.   Agent lookup and connection communication time.

Remote method calls are the heart of distributed object architectures such as CORBA, RMI, and DCOM. Socket communications, on the other hand, utilizes data streams for message passing. Somewhere in the middle is the Java Shared Data Toolkit. JSDT uti-

lizes socket communications to transport Data objects to the *dataReceived* method of the message recipient.

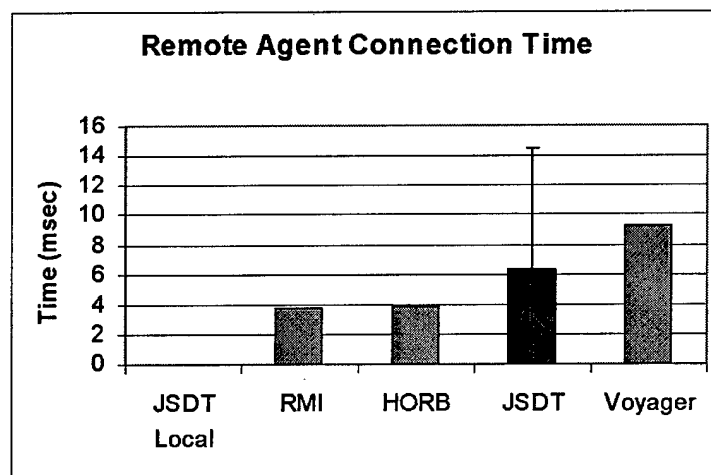In order to compare these methodologies, remote method calls are simulated using JSDT and sockets by sending the method arguments, operating on them at the receive end, and then sending the return value as a reply message (HYI98). The experiment uses three integer arguments and a single integer result.

The performance of JSDT compared with the results from (HYI98) can be seen in Figure 69. The JSDT simulated method calls perform an order of magnitude worse than Voyager agents, which themselves cost almost five times more than a C socket implementation. The JSDT *sendToClient* call addresses a recipient by name. Therefore, to complete the actual delivery, JSDT must perform a lookup of the target name within the list of the channel subscribers in order to get a pointer to the recipient agent. In essence embedded within a JSDT send operation is also a Registry naming lookup. This results in an increased send time. The reply send operates similarly. Additionally, all messages are sent as Data objects, which are larger and contain many fields above and beyond the simple integer primitive types. The result is an extremely lengthy send time. However, where JSDT fails miserably in speed, it excels in functionality.

In JSDT, send operations are inherently multithreaded and asynchronous. This is not the case with all the other methodologies. All other systems must busy wait for the remote method return value before proceeding. Additionally, all the architectures require obtaining a pointer or handle to the receiving agent before executing this remote method. Because of JSDT's send by name convention, references do not need to be obtained if agent names are known *a priori*. Furthermore, a multicast send requires no recipient references at all. All channel subscribers automatically receive the data. Finally, due to the asynchronous operation, agents can send their data and then perform additional processing. So, even though the send itself takes longer than other methodologies, it can be performed by a separate thread in the background, resulting in lower agent communication to computation time ratio. But, the multithreaded, asynchronous, operation represents a task that the processor must run concurrently. This can possibly reduce the system scaleability.

147

Remote Method Call Time
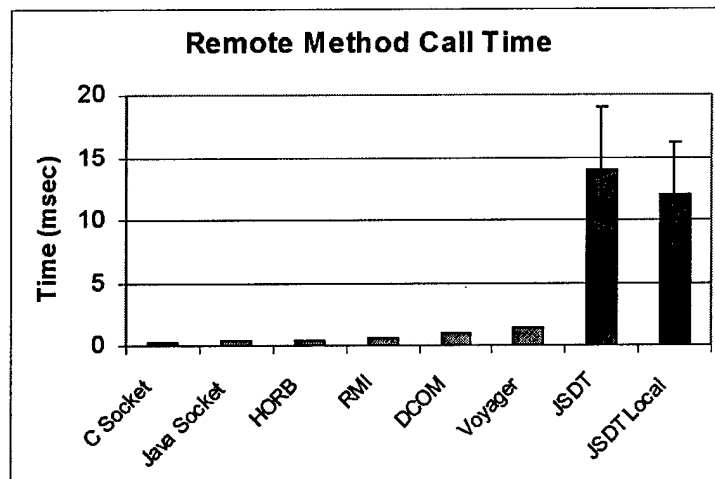
Figure 69.   Remote method call communication time.

To understand the effects of JSDT delivered agent messaging, tests are performed between a Controller agent and a varying number of Monitors conducting multiple simultaneous cStatus conversations. For this test, each Monitor sequentially generates 500 conversation threads. The Controller is hosted on ABC-A1, a dual CPU machine (Section 5.1.6, Table 24). The data indicates that increasing the number of Monitors initiating a cStatus conversation with a single Controller results in no significant difference in the conversation response times (Figure 70). The Controller was able to easily keep up with the increased workload with this number of Monitors (Figure 71). The CPU loading increases linearly with the number of Monitor agents and even with 8 initiators, the dual CPU's are only at about 20% loading on average. At this growth rate, it would take approximately 37 Monitor agents in order to increase both CPU utilizations to 100%. Only at this point would the average response times increase and overall system performance decrease. This agent-based architecture utilizing JSDT for communication exhibits excellent scaleability. A single Controller is projected to be able to support over 30 Monitors engaged in continuous conversations; a scenario which is not entirely likely. An anti-virus system is largely event driven. The chain of communications occurs only on a somewhat rare event, such as a virus detection or a system vaccination. Therefore, large numbers of agents generating multiple simultaneous conversations is an event that could only occur during an epidemic in a large system deployment. An unlikely occurrence, but even this

148

initial prototype appears capable of supporting a large scale deployment under extreme conditions.
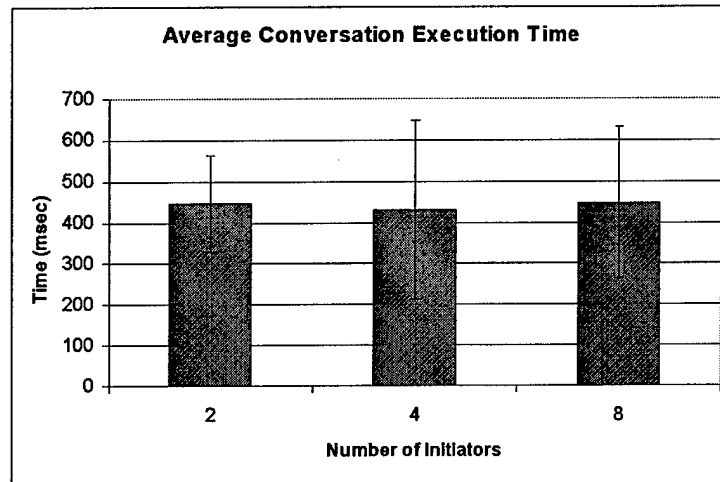
Figure 70.   Conversation response time during multiple simultaneous conversations.
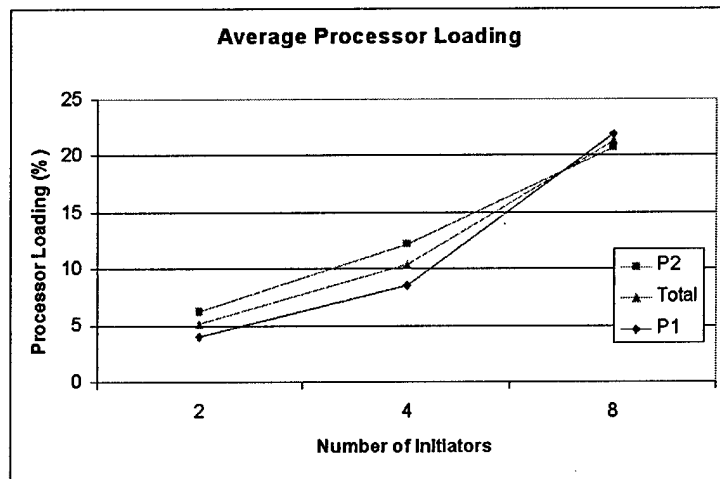
Figure 71.   Processor loading during multiple simultaneous conversations.

The JSDT environment offers a capable, effective, and scaleable communications backbone for this agent-based CVIS prototype. JSDT natively provides a registry, naming services, and asynchronous communications all utilizing the power of Java threads. However, JSDT is lacking in performance. The overhead required to provide the system flexibility and ease of use results in communication times that are significantly longer than

149

other methodologies. For this reason, other communication packages, such as CORBA could be investigated.

5.4 Summary

This chapter develops a thorough system test plan and discusses the results and analysis gained though system testing. The agent-based CVIS proves to be effective at detecting non-self within varying probabilistic error rates. The error rates are tunable through the proper selection of the number of fielded antibodies, the antibody length, and the detection threshold. The system is able to detect the presence of known *and* unknown malicious code, an advantage over current anti-virus solutions. Unfortunately, the performance limitations of Java affect the system efficiency and its usability in a real-world environment. However, the overall architecture did prove to be highly scaleable and even in its current implementation, would support an enterprise level deployment. These results and analysis lead to several conclusions across many of the integration domains. The conclusions are elaborated in the next chapter.

VI. Conclusions and Recommendations

The overall goal is to create an agent-based computer virus immune system. This was accomplished successfully though meeting the objectives. Effective system and local models of immune system operation were constructed that realize improvements over current anti-virus solutions. Based on these models, the multi-layered implementation provides an effective solution for the detection, identification, and elimination of computer viruses. The prototype was used to gain insight into the efficiency, effectiveness, and scaleability of an agent-based artificial immune system. The successful use of agents and the integration of pattern recognition principles are valuable contributions to the immunological computation community.

This research was conducted by integrating many different domains including immunology, immunological computation, malicious code, multiagent systems, and parallel & distributed computation. Because of the diverse amalgamation of ideas, conclusions are discussed from a variety of perspectives. These conclusions are based on the analysis of the design implementation.

6.1 Conclusions

System Models The system and local models for this CVIS are created based on ideas from biology, the self-adaptive CVIS (LMV99), the antibody lifecycle (HF99), and parallel computation. The separation of tasks into a logical hierarchy supports the reduction of the computational burden by allocating responsibilities to dedicated agents operating at the appropriate level. By integrating this structure with the prevention focus of the computer health system (CO99), a system-wide "computational health management" infrastructure is created that emphasizes preventative measures through information sharing. This infrastructure allows for the early identification and elimination of wide spread attacks, a feature missing from current AV capabilities. It also provides a forum for a collective self defense by enabling the sharing of successful antibodies among individual detectors in the "population." This diversity that is used to the advantage of the entire system is the result of the local model.

Each detector on each node within the system independently generates and manages its own antibody set. The computational burden on individual nodes is reduced by limiting the local number of antibodies. This distributes the cost of generation and negative selection across the system. These tasks can also be performed in parallel. Even though the detection capabilities at the local node are limited to the antibodies on hand, the full power of all the system scan strings can be utilized though information sharing via vaccinations. Additionally, each node is continually searching the non-self space through the "programmed cell death" within the local detector string lifecycle. This realizes the greatest advantage of this system over current methodologies, which is the ability to recognize as of yet unknown viral infections. The power gained through the partitioning of tasks and the sharing of information is accomplished through distributed, collaborating agents.

Agents The biological immune system is made up of many individual entities, each with their own "goals" and "services." Because of this, mapping the capabilities of these entities to software agents is an intuitive task. Additionally, the biological immune system components communicate through chemical signals. This elegantly maps to message passing in a distributed artificial immune system. For these reasons, the agent paradigm represents an excellent software engineering approach to AIS design.

The idea of agents does not represent a revolution in software design, but instead offers an extension to object-oriented software engineering that is useful for understanding and communicating about the components of complex, interacting systems. By discussing software objects in anthropomorphic terms, individuals are better able to visualize the system operation as related to everyday experience, such as one-to-one communication in a community of peers, using an expressive language. This is more intuitive than a client-server network utilizing several communications protocols. The agent abstraction provides an outstanding way to visualize and design the CVIS components based on their biological counterparts. However, below the top-level system view, the design and construction follow standard object-oriented approaches.

152

MaSE provides a superb software engineering approach that captures the nuances of agent-based design. However, the current object-oriented design tools do not interface well with the MaSE process. This results in a conceptual leap from the agent paradigm to object-oriented design after agents and their conversations are formulated. The standard MaSE methodology utilizes state transition diagrams in order to connect message receipt events to internal agent actions. However, STDs do not capture the temporal relationships in the conversation message passing protocols. The messages must arrive in the proper sequence for the conversation transaction to occur. Therefore, the MaSE methodology needs the addition of message sequence charts to fully specify conversations. Although they are not explicitly excluded, they are not included as part of the process and should be. Any further development on CASE tools based on the MaSE methodology (i.e. AgentTool) should include an environment for creating message sequence charts in order to properly define conversation protocols.

Antibodies The Detector agents each carry a battery of several antibody scan strings. In this prototype, these are generated pseudo-randomly. This provides a quick production method and because the exact locations of non-self within the search space are unknown, probably provides as good a method as any given all the possible non-self instantiations.

Testing shows that there exists an engineering trade-off between the specificity and generality of an antibody. Short strings are more general because they reduce the dimensionality of the self/non-self space and hence cover a larger area. A 4 byte antibody is shown to provide the coverage of a general detector string without the high negative selection cost of being too general. However, the cost of general detectors in terms of precision is unknown. Current AV solutions utilize 16 byte scan strings in order to help eliminate the threat of false positives. The CVIS accomplishes this through negative selection. However, short antibody lengths may not be able to adequately distinguish between self and non-self in cases where their differences are fine grained. The result is undetectable holes in the detector's ability to recognize non-self. This discussion alludes to a characterization of the self/non-self

space, which has not been accomplished for this problem domain. Future activities in this area could lead to the improved generation of antibodies through enhancing the random search by steering the generation algorithm towards known non-self areas of the search space.

Management Advantage Current anti-virus solutions are monolithic and provide little or no system wide management capabilities. Each desktop locally runs the complete AV package. All decisions for what and how to scan are left to the user. Even the addition of viral updates, vital to the continued effectiveness of the system, are often the task of the individual user to manually integrate. This prototype CVIS eliminates these problems and provides a framework for system metric reporting.

By using autonomous agents, this CVIS all but eliminates individual user interaction. Vaccinations and infection responses are controlled and directed by the agents at the network and system levels. Additionally, current system status is passed up the chain. This allows for automated metric collection, system status evaluation, and trend analysis. With the addition of appropriate logic, system-wide infection epidemics can be recognized and eliminated in real-time.

For the Air Force (and by extension the DoD) self is known and predefined by policy. Only approved applications are allowed on government information systems. From initialization on, the CVIS will find the presence of non-self (within a tunable probabilistic error rate). For the purposes of Air Force system custodians, this allows them to monitor for the presence of malicious code as well as non-approved application software.

Government agencies also have a reporting requirement for viral infections. Incident reports must be compiled and passed up the chain. With this architecture, virus incidents are already reported up the hierarchy. Automated incident report generation and statistics could be added to the metric generation duties of the Controller agents without much difficulty. This has the potential to save money and manpower that are currently being used to generate, report, and collate incident reports. Also available could be a real-time status display for infection incidents across the entire system. A live system status on malicious code incidents, similar to the network op-

erational status utilized by the GNOSC at DISA, could be generated and pictorially presented.

By integrating the ideas of an AV system hierarchy (MVL98) with the management and oversight processes of the public health system (CO99), this distributed agent-based CVIS provides a superior capability for system wide management and elimination of the virus threat over current solutions.

Issues There are several issues that remain unaddressed by this system including security and a time varying notion of self. No security layer is implemented in this prototype. The distributed nature of the system leaves it wide open to spoofing attacks that can compromise system integrity. Encrypted channels and digitally signed messages are required in order to ensure trusted conversations. JSDT can easily support these additions but what is ultimately required is a quality control mechanism for critical system components.

The system could be "trained" to generate an autoimmune reaction. By performing negative selection on non-self, all censored antibodies would react against self strings. These antibodies could then be passed on to other nodes using spoofed vaccination messages. The result would be false positive detections and the possible elimination of valid self applications. A trusted quality control mechanism is needed to oversee alarm generation and the dispensing of vaccinations.

Another problem is the constantly changing notion of self. Programs are continually added and deleted from most desktops. With the addition of an application, all antibodies would need to be re-censored against the new self additions. The problem with this methodology is the assumption that the new software is not infected. Alternatively, the new program can be scanned. A positive detection could indicate the presence of an infection or, the simple recognition that this "self" has not been encountered before by the system, a false positive. This problem is currently accounted for by the antibody life-cycle (Section 3.4.4, Figure 29). Any match requires a costimulation signal (currently from the system administrator) in order for an action to be taken. Therefore, the decision on whether this is a false positive or not rests with the judgment of the system administrator. While this scenario is accounted for, the

solution does not provide an assurance of detection and elimination, features core to the system effectiveness.

6.2 Implementation Changes to the Original Design

The system design represents a complete architecture for a CVIS. However, during implementation, several changes to this design were made to improve system operation, reduce unnecessary prototype complexity, and reduce schedule risk. The changes affect one agent, several conversations, and the conversation methodology.

The Complement agent is not present in the prototype implementation. The idea of a fast, less constrained scanning operation remains sound and is validated by experimental results (Section 5.2.3). However, issues in the areas of antibody coherency and coordination are raised between the Complement agents and their associated Detectors. The Complement agent offers probable improved system performance for scan operations. This is beyond the scope of the initial prototype, but could be included as part of future algorithmic improvements. The associated conversation to pass possible infections from a Complement agent to a Detector is also absent.

The cCostimulation conversation was originally designed to be initiated within other conversations in the raise alarm communications chain. During implementation, this proved to be awkward and inconsistent with other conversation designs. Therefore, it was broken up into separate cGraduateToMemory and cDestroyDetectorString conversations. These are now called by a Monitor agent in response to a positive, or negative, alarm costimulation.

Finally, implementation problems with the JSDT invite model resulted in a design change to the agent communication methodology. The socket-based agentMOM requires the use of a hailing channel in order to invite other agents to join a conversation (Section 4.2.4.2, Figure 44). The proposed design calls for using JSDT's ability to invite other clients through event services. This results in a much cleaner design (Section 4.2.8, Figure 47). However, creating the infrastructure for channel invitations requires creating channel managers and handling channel events. Although conceptually more elegant, the

architecture was unrealizable due to unresolved problems. Therefore, the final implementation utilizes the hailing channel methodology proposed by agentMOM. All agents join the hailing channel during registration with their appropriate session.

In spite of these changes, the remaining components are implemented as designed. The system is able to fully achieve the capabilities outlined in the system and local models. The extensive work initially expended on the system design has ensured the creation of an effective agent-based CVIS.

6.3   Future Research

Improved Scanning and Negative Selection Speed The current system can produce naive antibodies in 1.45 years for an 8GB drive and scan that drive in 1.05 years. This prototype system efficiency needs to be improved in order to be operationally viable.

Use of Compiled Languages C/C++ file access and scanning routines could be integrated into the current system through the use of the Java Native Interface (JNI). This functionality allows the calling of compiled C routines from a Java application. The system speed could be improved though very little effort via this route, while maintaining the current system architecture and communications functionality.

Parallel Censoring The prototype algorithm generates and performs negative selection sequentially. The algorithm execution time can be greatly reduced by generating an excess number of antibodies and then censoring them all in parallel. During negative selection, those antibody strings matching self are removed from the candidate population. After censoring, only naive strings remain. A sufficiently large number must be initially generated in order to ensure that enough remain after negative selection. This number of initial candidates must be estimated based on the antibody length, contents of self, detection threshold, and the number of remaining strings required after negative selection.

Efficient String Matching Improved methods of string pattern matching could be integrated to increase the performance of the matching algorithm. A common method

used in spell checking is to *a priori* construct a directed graph of the patterns. This is then used to process the input string against all patterns in a single pass by "walking" the graph (AC75).

Improved Matching Function The Rogers and Tanimoto measure along with 4 byte antibodies results in a detection value density distribution that approximates, but is not quite the same as, the ideal case. The use of 2 byte antibodies even more closely resembles the ideal case. However these are so general that they induce an extremely high negative selection time. Future research could create a matching function that produces the desired distribution, but uses a larger number of bytes in the antibody string. An improved function could be evolved through the use of genetic programming.

Antibody Creation The prototype uses a pseudo-random number generator to create antibody candidates. These are then censored at a very high rate to produce valid detection strings. Improved antibody generation schemes could reduce the censoring rate by directing the creation algorithm to known areas of the non-self space. This would improve the generation and negative selection efficiency.

Affinity Maturation The current implementation of deploying randomly generated antibodies can result in multiple matches on the same antigen. Affinity maturation could be implemented to conserve resources by only retaining the antibody with the highest affinity. This could be extended to include hypermutation and clonal selection algorithms to create evolved copies of high affinity antibodies. This has the possibility of improving the system adaption process and also increasing the detection of related viral strains.

Metrics One of the goals of the Controller agent is to produce metrics on system performance (Section 4.1.1). This functionality is necessary for management insight into system operation and in order to understand the system wide impact of viruses. Real time displays could also be created based on the metric information. This functionality is not currently not implemented.

Additional Detectors Only file infector viruses are detected with the prototype system. Additional agent types need to be created in order to detect and remove the other viral threats, such as macro and boot sector viruses (Section 2.1.1). A complete set of detector types is required to create a multi-layered defense in depth.

User Interfaces This prototype offers a command-line only functionality with little access to system operation, except for test scripts. Graphical user interfaces (GUIs) need to be created in order to interface more effectively, and efficiently, with the CVIS agents. This would also allow for more user control over system variables, such as scan location, which are currently hard coded.

Robust Deployment The prototype contains very little code to deal with system failures. However, the system architecture is designed to one-day accommodate such functionality. Features should be added to support the graceful degradation of service in the face of failure, instead of system collapse. This could include backup agents, such as Monitors that automatically fail over to their adjacent peers, and communications timeouts with recovery.

Security The current system is highly vulnerable to spoofing and denial of service attacks. For instance, erroneous vaccinations could easily be sent to a Detector, which could cause an autoimmune reaction. The prototype architecture easily supports the addition of security layers, such as secure socket communication and agent authentication, but they are not currently implemented. These would have to be added, especially for a wide area network deployment, in order to overcome the security problems associated with system compromise.

6.4 Summary

The system design integrates the power, flexibility, adaption, and capabilities of the biological immune system into an architecture realizable in the information system domain. Based on the models, the prototype implementation provides an effective solution for the detection, identification, and elimination of malicious code. The level of effectiveness is tunable through the proper selection of the number of antibodies, the antibody length,

and the detection threshold. These must be selected based on the contents of known self and with an understanding of their ramifications on negative selection time, scan time, and non-self space coverage.

The use of the agent paradigm facilitates the construction of an artificial immune system because of the performance limitations of a monolithic implementation and the biological basis for the architecture can be viewed as a system of collaborating agents (Syc98). While using agents improves the understanding of the system design and the mapping to the biological domain, the deployment of the agents must be done by considering the principles of parallel software design in order to improve performance. For an agent-based CVIS, this involves reducing communication and placing detection agents near their I/O sources.

This CVIS design is scaleable in terms of scope and coverage through the simple addition of new agent types and participating system nodes. The prototype only implements file system detection, but a more complete multi-layered defense could be realized by adding agent types for monitoring memory, email, and boot sectors. Additionally, because the Java Shared Data Toolkit provides lookup services, agents can join or leave the system at anytime.

Testing reveals that the communication services provided by JSDT are scaleable up to about 37 simultaneous conversations using midrange server hardware (ABC-A1, Section 5.1.6, Table 24). Using only one Controller agent (an improper deployment due to the single point of failure) and a pyramid deployment of nodes underneath it, over 1400 agents engaged in constant communication can be supported. Even this initial prototype provides enough scaleability for an enterprise wide deployment. Implementation and algorithmic improvements could expand this.

While this effort proved to be effective and scaleable, the prototype performance does not provide for a practical implementation nor unobtrusive operation. The communication times of the JSDT are disappointing and although they are executed as background tasks, their speed indicates that alternate communication solutions should be investigated for an actual fielded implementation. The Java implementation provides a good prototype

environment, but its speed limits the system usability. The negative selection and scanning times measured in years are unacceptable for a practical system. An implementation improvement to increase the system speed is paramount to future system viability.

The agent-based computer virus immune system offers detection and management capabilities that are absent from current solutions. These facets are able to work together to provide enterprise wide viral protection. This research focuses on the computer virus problem domain, but artificial immune systems can be applied to other areas as well. An obvious extension would be network intrusion detection (by changing the scan string composition), but other uses include machine learning and function optimization. The effectiveness of distributed problem solving through an agent-based approach demonstrated here could be applied to other immune system projects to improve their performance.

List of Acronyms

ADK - Agent Development Kit.

AgML - Agent Modeling Language.

AIS - Artificial Immune System.

APC - Antigen Presenting Cell.

API - Application Program Interface.

ARG - Agent Research Group.

AV - Anti-Virus.

BIS - Biological Immune System.

C4ISR - Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance.

CHS - Computer Health System.

COM - Component Object Model.

CORBA - Component Object Request Broker Architecture.

CPU - Central Processing Unit.

CRC - Cyclic Redundancy Code.

CVIS - Computer Virus Immune System.

CVC - Center for Virus Control.

DCOM - Distributed Component Object Model.

DNA - Deoxyribose Nucleic Acid.

DOS - Disk Operating System.

EICAR - European Institute for Computer Anti-Virus Research.

GA - Genetic Algorithm.

GUID - Global Unique Identifier.

HTTP - Hypertext Transfer Protocol.

JIT - Just In Time (Compiler).

JMQ - Java Message Queue.

JMS - Java Message Service.

JNI - Java Native Interface.

JSDT - Java Shared Data Toolkit.

JVM - Java Virtual Machine.

LRMP - Light-weight Reliable Multicast Package.

MAS - Multi-agent System.

MaSE - Multi-agent Systems Engineering.

MBR - Master Boot Record.

MHC - Major Histocompatibility Complex.

MOM - Message Oriented Middleware.

MOP - Measure of Performance.

MPI - Message Passing Interface.

NAV - Norton Anti-Virus.

OO - Object-oriented.

OOA - Object-oriented Analysis.

OOD - Object-oriented Design.

RMI - Remote Method Invocation.

RPC - Remote Procedure Call.

SNR - Signal to Noise Ratio.

SSL - Secure Socket Layer.

STD - State Transition Diagram.

TSR - Terminate and Stay Resident.

UML - Unified Modeling Language.

UNM - University of New Mexico.

VBA - Visual Basic for Applications.

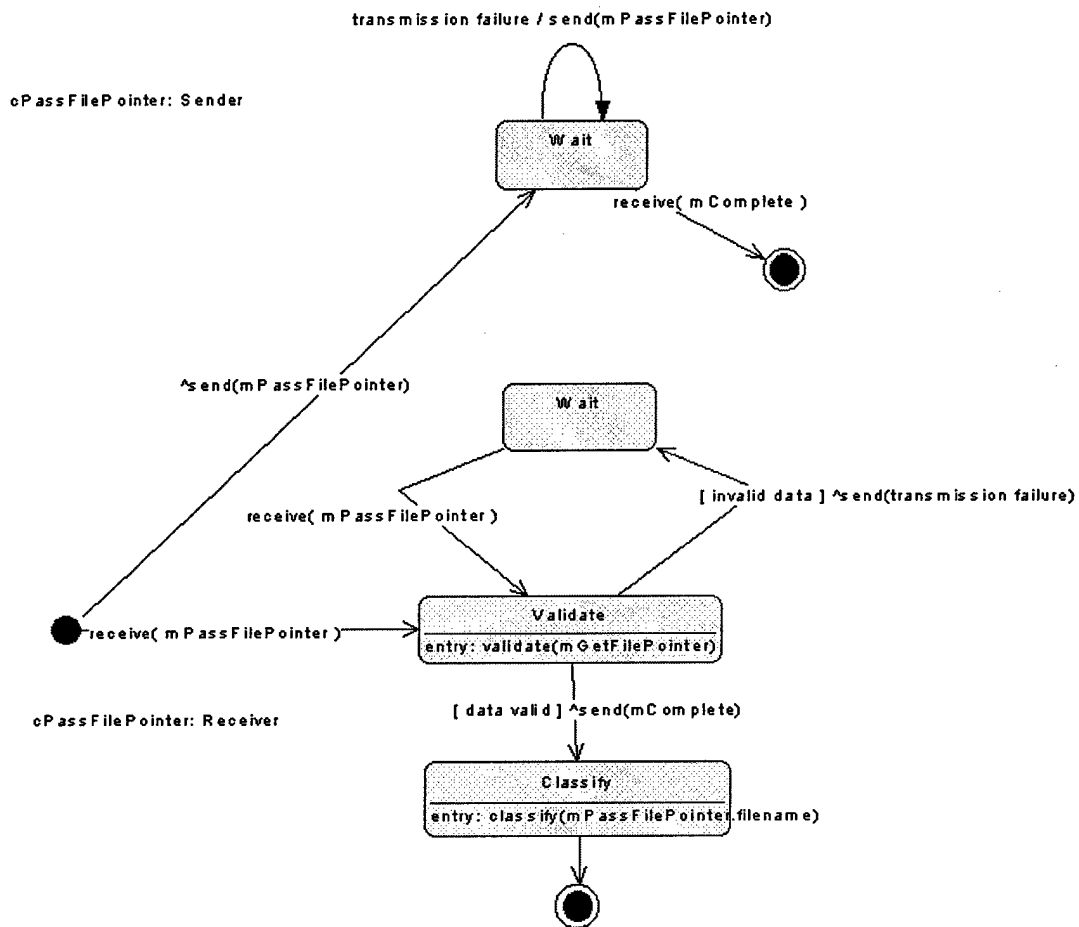**VPA** - Virus Prevention Agency.

WM - Word Macro (virus). e.g. WM.Concept.

XM - Excel Macro (virus). e.g XM.Laroux.

Appendix A. Design Documentation

A.1 Agent Conversations

A.1.1 State Transition Diagrams.

transmission failure / send(mPassFilePointer)

cPassFilePointer: Sender

W ait

receive(mComplete)

^send(mPassFilePointer)

W ait

receive(mPassFilePointer)

[invalid data] ^send(transmission failure)

receive(mPassFilePointer)

Validate

entry: validate(mGetFilePointer)

cPassFilePointer: Receiver

[data valid] ^send(mComplete)

Classify

entry: classify(mPassFilePointer.filename)

Pass file pointer conversation.

cRaiseAlarm : Sender

Separate threads.
All must complete
to reach end state.

transmission failure ^send(m RaiseAlarm)

receive(m Complete)[agent:instanceOfMonitor]

receive(m ValidAlarm) receive(m InvalidAlarm)

Wait

oCostimulation

^send(m RaiseAlarm)

cRaiseAlarm : Receiver

receive(m RaiseAlarm)

Wait2

[invalid data]^send(transmission failure)

[valid data, receiver is Monitor]^send(m Complete)

[valid data, receiver is a Helper]

receive(m RaiseAlarm)

ValidateRaiseAlarm
entry:validate(m RaiseAlarm)

oUpdateThreshold

[positive costimulation]^send(m ValidAlarm)

[negative costimulation]^send(m InvalidAlarm)

Costimulation
entry:getUserInput(costimulation)

Raise an alarm conversation.

cVaccination: Sender

transmission failure ^send(mVaccination)

Wait

send(mVaccination)

receive(mComplete)

[valid data, agent instanceof Monitor] ^send(mVaccination)

cVaccination: Receiver

Wait2

[invalid data] ^send(transmission failure)

receive(mVaccination)

receive(mVaccination)

ValidateVaccination

entry: validate(mVaccination)

[valid data, agent instanceof Detector] ^send(mComplete)
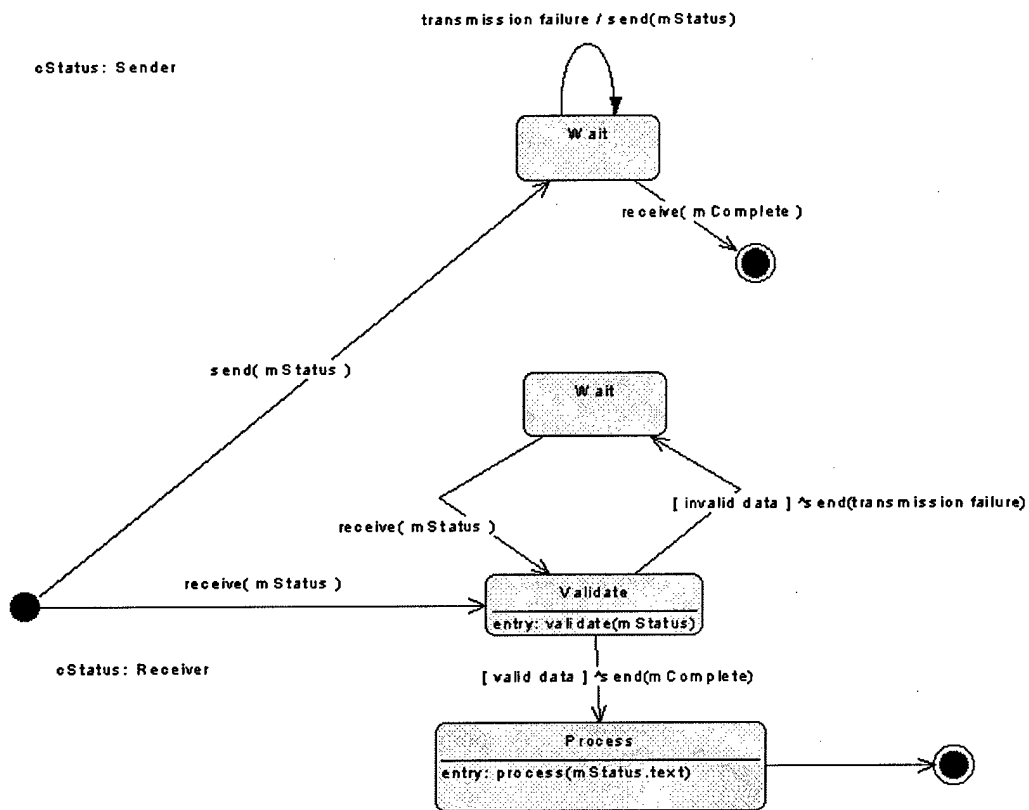
ProcessVaccination

entry: vaccinate(mVaccination.pattern)

Vaccination conversation.
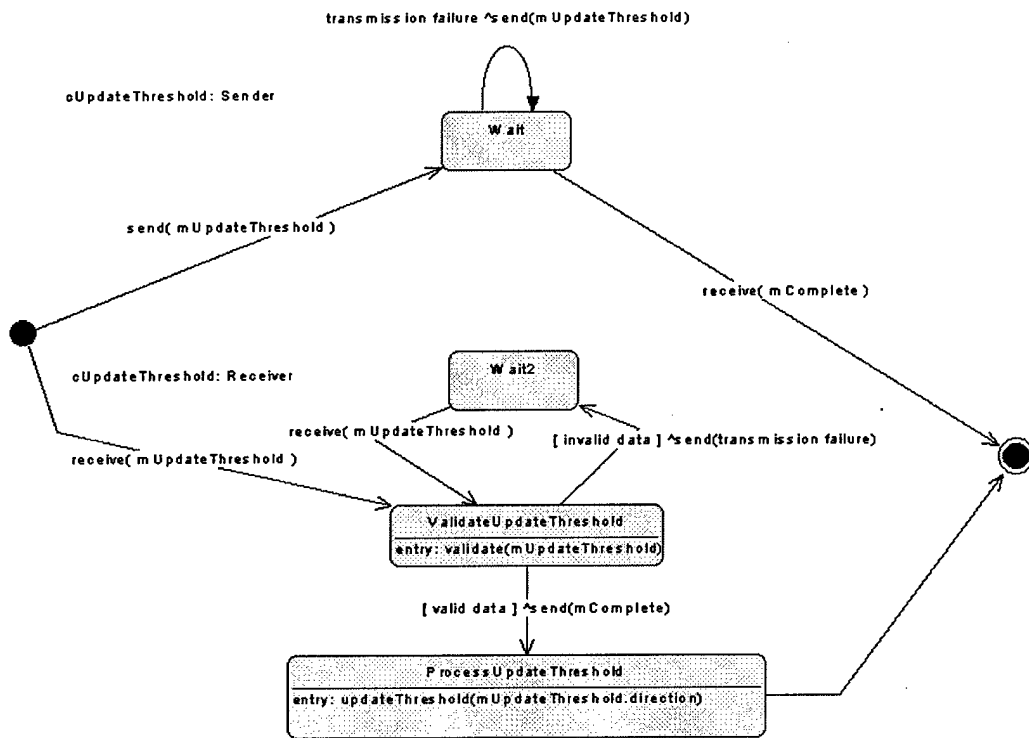
Raise a warning conversation.
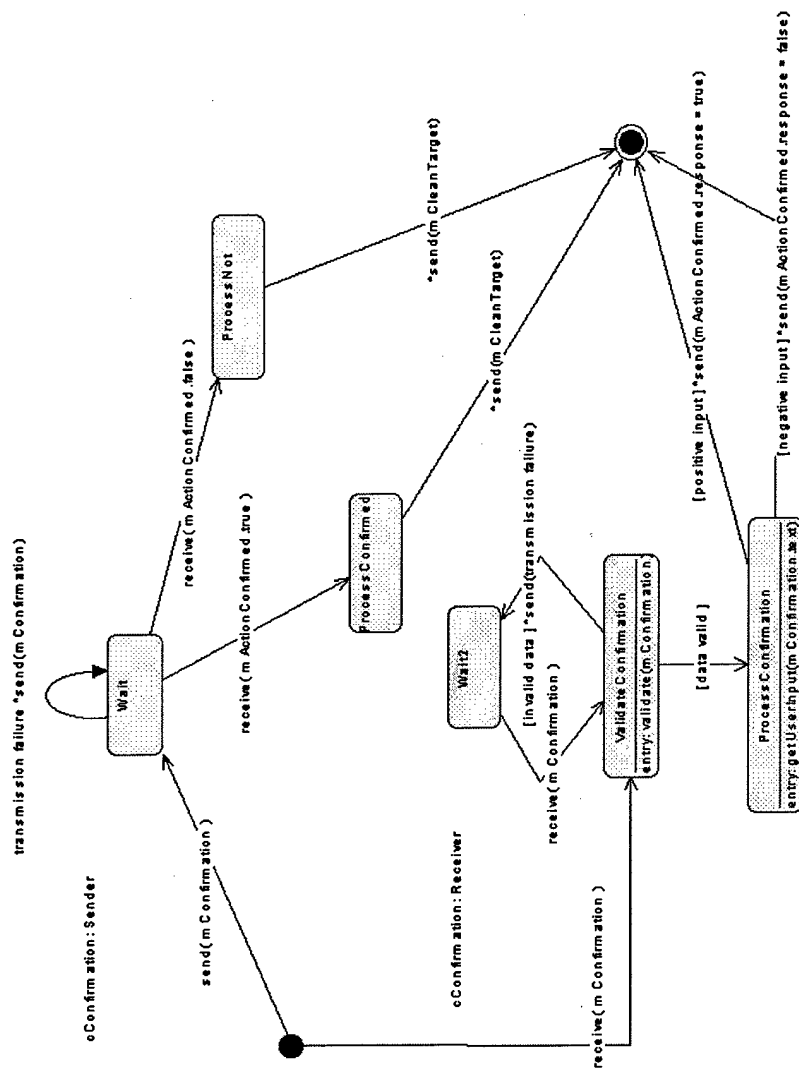
Action confirmation conversation.

Agent or system status passing conversation.
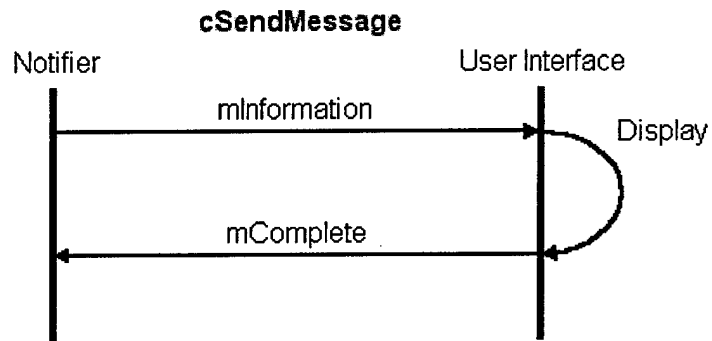
transmiss ion failure ^s end(m UpdateThres hold)

oUpdate Thres hold: Sen der

W ait

s end(m UpdateThres hold)

receive(m C omplete)

oUpdate Thres hold: R eceiver

W ait2

receive(m UpdateThres hold)

[invalid d ata] ^s end(trans miss ion failure)

receive(m UpdateThres hold)

V alidateU pdateThres hold

entry: validate(m UpdateThres hold)

[valid d ata] ^s end(m C omplete)

P rocessU pdate Thres hold

entry: updateThres hold(m U pdate Thres hold.direction)

Update matching threshold conversation.

Agent message passing conversation.

Infection costimulation conversation.

173

A.1.2 Message Sequence Charts.

cSendMessage

Notifier   User Interface

mInformation

Display

mComplete

Agent message passing conversation.

cStatus

Sender   Receiver

mStatus

Process

mComplete

Agent or system status passing conversation.

cUpdateThreshold

Sender   Implementer

mUpdateThreshold

Process

mComplete

Update matching threshold conversation.

cVaccination

Vaccination conversation.

cPassFilePointer

Pass file pointer conversation.

Agent alarm notification and resolution conversation chain.

176

A.2 Conversation Messages

Agent conversation messages.
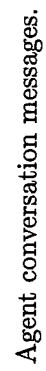
A.3 Agent System Design

Agent object diagram.

178
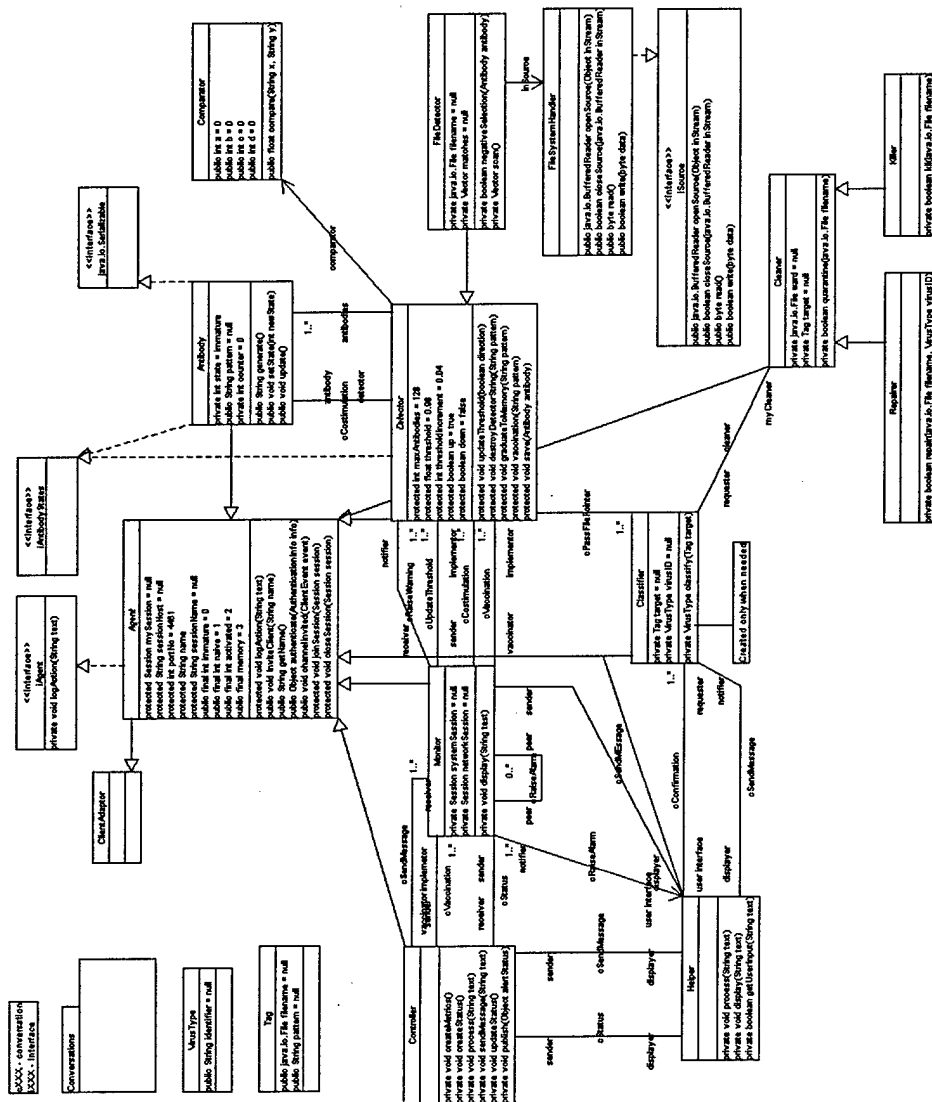
Appendix B. Source Code Availability

The source code and original design documentation for the agent-based computer virus immune system is not included as part of this document. Those interested in obtaining a copy of these should direct their requests to:

Dr. Gary Lamont
AFIT/ENG
2950 P Street
WPAFB, OH 45433-7765

gary.lamont@afit.af.mil

Bibliography

AC75.   Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

AL98.   Yariv Aridor and Danny B. Lange. Agent design patterns: Elements of agent application design. *Proceedings of Autonomous Agents 1998*, pages 108–115, 1998.

Bac96.   Thomas Back. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

Bak98.   Mark Baker. mpiJava: A Java MPI Interface. Workshop presented at the European Parallel Conference (EuroPar), September 1998.

BB98.   Joseph P. Bigus and Jennifer Bigus. *Constructing Intelligent Agents with Java: A Programmer's Guide to Smarter Applications*. Wiley Computer Publishing, New York, 1998.

Bon94.   Vesselin Bontchev. *Future Trends in Virus Writing*. Virus Test Center, University of Hamburg, Hamburg, Germany, April 1994.

BP78.   Patricia J. Blake and Rosanne C. Perez. *Applied Immunological Concepts*. Appleton-Century-Crofts, New York, 1978.

Bra97.   Jeffrey M. Bradshaw, editor. *Software Agents*. The MIT Press, Menlo Park, CA, 1997.

Bre98.   Walter et al. Brenner. *Intelligent Software Agents: Foundations and Applications*. Springer, Berlin, 1998.

BSL96.   Eli Benjamini, Geoffrey Sunshine, and Sidney Leskowitz. *Immunology: A Short Course*. Wiley-Liss, Inc., New York, third edition, 1996.

Bur99.   Rich Burridge. *Java Shared Data Toolkit User Guide*. Sun Microsystems, Inc., Mountain View, CA, April 1999. Version 1.5.

CDM79.   Harlan Crowder, Ron S. Dembo, and John M. Mulvey. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software*, 5(2):193–203, June 1979.

CFKL99. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava 1.2:API Specification. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html, October 1999.

Cha97.   Deepika Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, University of Cincinnati, 1997.

CO99.   Kelley J. Cardinale and Hugh M. O'Donnell. A constructive induction approach to computer immunology. Master's thesis, Air Force Institute of Technology, WPAFB, OH, March 1999. AFIT/GCS/ENG/99M-02.

Coh94.   Frederick B. Cohen. *A Short Course on Computer Viruses*. John Wiley & Sons, Inc., New York, second edition, 1994.

Das98.   Dipankar Dasgupta. An artificial immune system as a multi-agent decision support system. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 3816–3820, October 1998.

Das99.   Dipankar Dasgupta, editor. *Artificial Immune Systems and Their Applications*. Springer, Heidelberg, Germany, 1999.

DD98.   H. M. Deitel and P. J. Deitel. *Java How to Program*. Prentice Hall, Upper Saddle River, NJ, second edition, 1998.

DeL99a.   Scott A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems. *Proceedings of the International Bi-Conference Workshop on Agent-Oriented Information Systems*, May 1999.

DeL99b.   Scott A. DeLoach. Using agentMOM. Software users manual, 1999.

DFH96.   Patrik D'haeseleer, Stephanie Forrest, and Paul Helman. An immunological approach to change detection: Algorithms, analysis and implications. *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

Duc99.   Paul Ducklin. Standard anti-virus test file. European Institute for Computer Anti-Virus Research, August 1999.

FAPC94.   Stephanie Forrest, Lawrence Allen, Alan S. Perelson, and Rajesh Cherukuri. Self-nonself discrimination in a computer. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1994.

FG96.   Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, pages 21–36, April 1996.

FH99.   Eric Freeman and Susanne Hupfer. Make room for JavaSpaces, part 1. *Sun World*, November 1999.

FHA99.   Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Jini Technology. Addison-Wesley, Reading, MA, June 1999.

FHS97.   Stephanie Forrest, Steven A. Hofmeyer, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

FHSL96.   Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, pages 120–128, 1996.

For97.   Richard Ford. Why viruses are and always will be a problem. http://www.icsa.net/library/research/d.shtml, September 1997.

Fou93.   Mayo Foundation. Mayo clinic family health book. Computer Software, 1993. Version 1.2.

FSJP93. Stephanie Forrest, Robert E. Smith, Brenda Javornik, and Alan S. Perelson. Using genetic algorithms to explore pattern recognition in the immune system. *Evolutionary Computation*, 1(3):191–211, July 1993.

Gic98. Patricia C. Giclas. *Diagnostic Immunology and Complement Laboratory Catalog.* National Jewish Medical and Research Center, Denver, CO, December 1998.

GK95. Michael R. Genesereth and Steven P. Ketchpel. Software agents. March 1995.

Gor93. David S. Gordon. Encarta. Computer Software, 1993.

Hal98. Stephen S. Hall. *Arousing the Fury of the Immue System.* Howard Hughes Medical Institute, Chevy Chase, Maryland, 1998.

HC96. John Hunt and Denise Cooke. The ISYS project: An introduction. Technical Report IP-REP-002, University of Wales, Aberystwyth, Penglais, Aberystwyth, Dyfed, UK, March 1996.

HCK95. Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Research report, IBM Research Division, Yorktown Heights, NY, March 1995.

Her96. Her Majesty's Office of Information. Antibodies teach computers to learn. http://www.aber.ac.uk/ jot/ISYS/hmoi.html, September 1996. Press Release.

HF99. Steven A. Hofmeyr and Stephanie Forrest. Immunity by design: An artificial immune system. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1289 – 1296, July 1999.

HJTW99. Daniela E. Herlea, Catholijn M. Jonker, Jan Treur, and Niek J.E. Wijngaards. Specification of behavioural requirements within compositional multi-agent system design. *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1999.

Hof90. Lance J. Hoffman, editor. *Rogue Programs: Viruses, Worms, and Trojan Horses.* Van Nostrand Reinhold, New York, 1990.

Hof97. Steven A. Hofmeyr. An overview of the immune system. http://www.cs.unm.edu/ immsec/html-imm/immune-system.html, 1997.

HRN98. Emma Hart, Peter Ross, and Jeremy Nelson. Producing robust schedules via an artificial immune system. *Proceedings of the IEEE International Conference on Evolutionary Computing*, May 1998.

HYI98. Satoshi Hirano, Yoshiji Yasu, and Hirotaka Igarashi. Performance evaluation of popular distributed object technologies for java. *Proceedings of the ACM Workshop on High-Performance Network Computing for Java.*, March 1998.

Int98. International Business Machines Inc. Understanding viruses. http://www.av.ibm.com/InsideTheLab/Bookshelf/Understanding/, 1998.

Jan93. Charles A. Janeway, Jr. How the immune system recognizes invaders. *Scientific American*, pages 73–79, September 1993.

KA94.   Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. *4th Virus Bulletin International Conference,* pages 179–194, 1994. R. Ford, editor.

Kep94.   Jeffrey O. Kephart. A biologically inspired immune system for computers. *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems,* pages 130–139, 1994.

KGGK94.   Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

KSSW97.   Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer, and Steve R. White. Blueprint for a computer immune system. *Proceedings of the Virus Bulletin International Conference,* 1997.

KT98.   Neeran M. Karnik and Anand R. Tripathi. Design issues in mobile-agent programming systems. *IEEE Concurrency,* pages 52–61, July-September 1998.

Lad97.   Mark D. Ladue. When java was one: Threats from hostile byte code. *Proceedings of the 20th Annual National Information Systems Security Conference,* pages 104–115, October 1997.

Leo00.   Mark Leon. Internet virus boom. *Infoworld,* 22(3):36–37, January 2000.

LMV99.   Gary B. Lamont, Robert E. Marmelstein, and David A. Van Veldhuizen. *New Ideas in Optimization,* chapter A Distributed Architecture for a Self-Adaptive Computer Virus Immune System, pages 167–183. Advanced Topics in Computer Science Series. McGraw-Hill, London, 1999.

Lud96.   Mark A Ludwig. *The Little Black Book of Computer Viruses.* American Eagle Publications, Inc., Show Low, Arizona, 1996.

Mah00.   Qusay H. Mahmoud. *Distributed Programming with Java.* Manning, Greenwich, CT, 2000.

MCD99.   Bill McCarty and Luke Cassady-Dorion. *Java Distributed Objects: The Authoritative Solution.* SAMS, Indianapolis, Indiana, 1999.

MTF96.   Kazuyuki Mori, Makoto Tsukiyama, and Toyoo Fukuda. Multi-optimization by immune algorithm with diversity and learning. *Proceedings of the Second International Conference on Multiagent Systems,* pages 118–123, December 1996.

MVHL99.   Robert E. Marmelstein, David A. Van Veldhuizen, Paul K. Harmer, and Gary B. Lamont. A white paper on modeling & analysis of computer immune systems using evolutionary algorithms. TR 1, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1999.

MVL98.   Robert E. Marmelstein, David A. Van Veldhuizen, and Gary B. Lamont. A distributed architecture for an adaptive computer virus immune system.

Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC), October 1998.

Nat97.   National Computer Security Association. *NCSA 1997 Computer Virus Prevalence Survey*, 1997.

Nat99.   National Computer Security Association. *NCSA 1999 Computer Virus Prevalence Survey*, 1999.

Net98.   Network Associates Inc. *The Growth of the Virus Threat*, 1998. http://www.drsolomon.com/vircen/stats.cfm.

NPc99.   A compendium of NP optimization problems. http://www.nada.kth.se/ viggo/problemlist/ compendium.html, May 1999.

NS93.   Morton Nadler and Eric P. Smith. *Pattern Recognition Engineering*. John Wiley & Sons Inc., New York, 1993.

Pre94.   Richard Preston. *The Hot Zone*. Doubleday, New York, 1994.

Pre97.   Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, fourth edition, 1997.

Ref99.   Tally's Virii Link Reference. Tallys virus collection statistics. http://www.virusexchange.org/tally/stats1.html, November 1999.

Ret99.   Reticular Systems, Inc. Agent construction tools. http://www.agentbuilder.com/AgentTools/index.html, June 1999.

RLC$^+$99. Rajeev R. Raje, Zhiqing Liu, Sivakumar Chinnasamy, Joseph Williams, Wilfred Mascarenhas, and Ming Zhong. *High Performance Cluster Computing*, chapter 12, pages 249–273. 1999.

RN95.   Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

Sch99.   Douglas C. Schmidt. The adaptive communication environment (ACE). http://www.cs.wustl.edu/ schmidt/ACE.html, December 1999.

SHF97.   Anil Somayaji, Steven Hofmeyer, and Stephanie Forrest. Principles of a computer immune system. *Proceedings New Security Paradigms '97*, 1997.

Ska96.   Rune Skardhamar. *Virus Detection And Elimination*. AP Professional, New York, 1996.

Sla96.   Robert Slade. *Robert Slade's Guide to Computer Viruses*. Springer-Verlag, New York, second edition, 1996.

Smi99.   Jerry Smith. Distributed computing with aglets. http://www.vistabonita.com/papers/DCAglets/DCWithAglets.html, 1999.

SOHL$^+$96. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1996.

184

Som97.   Bret Sommers. Agents: Not just for bond anymore. *JavaWorld*, April 1997.

Sun98.   Todd Sundsted. An introduction to agents. *JavaWorld*, June 1998.

Sun99a.   Sun Microsystems, Inc. Java Message Queue Features & Benefits.
            http://www.sun.com/workshop/jmq/features.html, November 1999.

Sun99b.   Sun Microsystems, Inc. Java message queue quickstart guide.
            http://www.sun.com/workshop/jmq/, October 1999. Beta Draft Version.

Sun99c.   Sun Microsystems, Inc. Java message service.
            http://www.javasoft.com/products/jms/, 1999. Version 1.0.2.

Syc98.   Katia P. Sycara. Multiagent systems. *AI Magazine*, 19(2):79–92, 1998.

Sym00.   Symantec. Norton antivirus for windows 95. Computer Software, 2000.
            Version 95.0.b.

TG74.   Julius T. Tou and Rafael C. Gonzalez. *Pattern Recognition Principles*.
            Addison-Wesley, Reading, MA, 1974.

TS91.   Walter A. Triebel and Avtar Singh. *The 8088 and 8086 Microprocessors:
            Programming, Interfacing, Software, Hardware, and Applications*. Prentice
            Hall, Englewood Clifts, NJ, 1991.

Ven97.   Bill Venners. Under the hood: The architecture of aglets. *JavaWorld*, April
            1997.

Wel96.   Joe Wells. The timeline.
            http://www.av.ibm.com/InsideTheLab/Bookshelf/Timeline/, August 1996.

Wel98.   Joe Wells. The wildlist. *Virus Bulletin*, October 1998.

WM97.   D. H. Wolpert and W. G. Macready. No free lunch theorems. *IEEE
            Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Vita

    1st Lt Paul Harmer, born in Pasadena, CA, graduated from California State University Long Beach with a Bachelor of Science degree in Electrical Engineering. He is a distinguished graduate of AFROTC, detachment 040, Loyola Marymount University, Los Angeles, CA. His first assignment was with the Space and Missile Systems Center, Satellite and Launch Control System Program Office at Vandenberg AFB, CA where he was in charge of acquisition, test, and transition of spacelift range modernizations. Upon graduation, Lt Harmer will be assigned to the Air Force Research Laboratory, Sensors Directorate, Wright-Patterson AFB, OH. He will be working on applying parallel computation resources to automatic target recognition problems. He is married to the former Briony Speth of San Diego, CA.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
A DISTRIBUTED AGENT ARCHITECTURE FOR A COMPUTER VIRUS IMMUNE SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Paul K. Harmer, 1st Lt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management
2950 P Street, Building 640
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/00M-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Captain Freeman Alex Kilpatrick
AFOSR/NM
801 North Randolph Street
Room 732 9-65
Arlington VA 22203-1977 (703) 696-6565

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Gary B. Lamont, ENG, DSN: 785-3636 x4718 COMM: (937)255-3636 x4718

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Information protection and information assurance are vital components required for achieving superiority in the Infosphere, but these goals are threatened by the exponential birth rate of new computer viruses. The increased global interconnectivity that is empowering advanced information systems is also increasing the spread of malicious code and current anti-virus solutions are quickly becoming overwhelmed by the burden of capturing and classifying new viral stains. To overcome this problem, a distributed computer virus immune system (CVIS) based on biological strategies is developed. This research develops a model of the biological immune system (BIS) and utilizes software agents to implement a CVIS. The system design and implementation validates that agents are an effective methodology for the construction of an artificial immune system. The distributed agent architecture provides support for detection and management capabilities that are unavailable in current anti-virus solutions. The detector agents are able to distinguish self from non-self within a probabilistic error rate that is tunable through the proper selection of system parameters. This research also shows that by fighting viruses using an immune system model, that known, and previously unknown, malicious code can be detected and removed from the system.

14. SUBJECT TERMS
Computer Viruses, Distributed Data Processing, Pattern Recognition, Artificial Immune Systems, Agent-oriented Information Systems

15. NUMBER OF PAGES
203

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL